معماری کامپیوتر (3واحد درسی تئوری)

تهیه کننده:مجتبی پورمحقق

این درس در مورد چیست؟

این درس در مورد ساختار وچگونگی طراحی کامپیوتر های دیجیتالی است . این مطلب معروف به "معماری کامپیوتر" است (که شامل معماری مجموعه دستورالعمل + سازماندهی سخت افزاری می باشد) .

چرا سازماندهی کامپیوتر را بررسی می کنیم

شاید هیچکدام از شما در شرکتهای Intel و یا AMD کار نکرده اید اما . . .

- کامپیوترهای جاسازی شده
- طراحی کامپایلر
- حتی طراحان نرم افزار

با محصولات این شرکتها در ارتباطند .

طرح کلي درس

- بازنگری کلی
- کارایی
- مجموعه دستورالعملها
- محاسبات کامپیوتر
- ماشینهای تک چرخه ای
- خط لوله ای
- سیستمهای حافظه (RAM , Caches , Virtual Memory)
- سوپر اسکالر (Superscalar/VLIW) و چند پردازنده ها
- مباحث دیگر

سرفصل 1: رئوس مطالب در معماری و سازماندهی کامپیوتر

- یافتن توانایی ارائه اطلاعات پایه از معماری و سازماندهی کامپیوتر در جریان طراحی کامل یک کامپیوتر

- دریافتن مسئولیتهای حرفه ای و اخلاقی یک مهندس کامپیوتر (مخصوصا معمار کامپیوتر)

”معماری“ به چه معناست؟

” فن یا دانش یا ساختمان ... فن یا پرداختن به طراحی و پیاده سازی ساختارها ... ”

- Webster 9th New College Dictionary

”شامل نقشه ، طراحی ، ساخته و دکوراسیون چگونگی عملکرد ”

- American College Dictionary

”معماری کامپیوتر“

- کلمه ای که توسط Fred Brooks ابداع گردید.

”معماری کامپیوتر“

”معماری کامپیوتر، یعنی کامپیوتر از دید کاربر“

- Amdhal et al, (64)

”ما بوسیله معماری، ساختارواحدهای تشکیل دهنده یک سیستم کامپیوتری را هدفمند می نمائیم .“

- Stone, H. (1987)

"معماری کامپیوتر"

"معماری یک کامپیوتر عبارتست ازمحیط یا فضای
بین ماشین و نرم افزار"

- Andris Padges

IBM 360/370 Architect

”معماری کامپیوتر“

ساختار: نظم و ترتیب دادن به بخشهای ثابت (نقشه)

سازماندهی: فعل و انفعال پویای این بخشها و مدیریت آنها

پیاده سازی: طراحی کردن بلوک بخشهای دارای هدف خاص

ارزیابی کارایی: مطالعه رفتار سیستم (decorative treatment)

معماری (از دیدگاه معمار)

- پیاده سازی
- سازماندهی: منظر سطح بالا
  - سیستم حافظه
  - ساختار گذرگاه (bus)
  - طراحی داخلی CPU
- سخت افزار
  - طراحی منطقی
  - تکنولوژی بسته بندی(packaging)
- معماری مجموعه دستورالعمل

نکات مهم

- به خاطر بسپارید: نکته اینست که بیاموزید چگونه معماری به مفهوم تکنولوژی موجود را ارزیابی کنید.
- شناختن روش خیلی مهم است، اما پایان کار نیست.

مراحل در سازماندهی کامپیوتر

- مفهوم ماشینهای چند سطحی
- مفاهیم ماشین مجازی

انضباط در معماری

- ساختار سخت افزار / نرم افزار
- الگوریتم ها و پیاده سازی آنها
- انتشار زبان

تصویر بزرگ

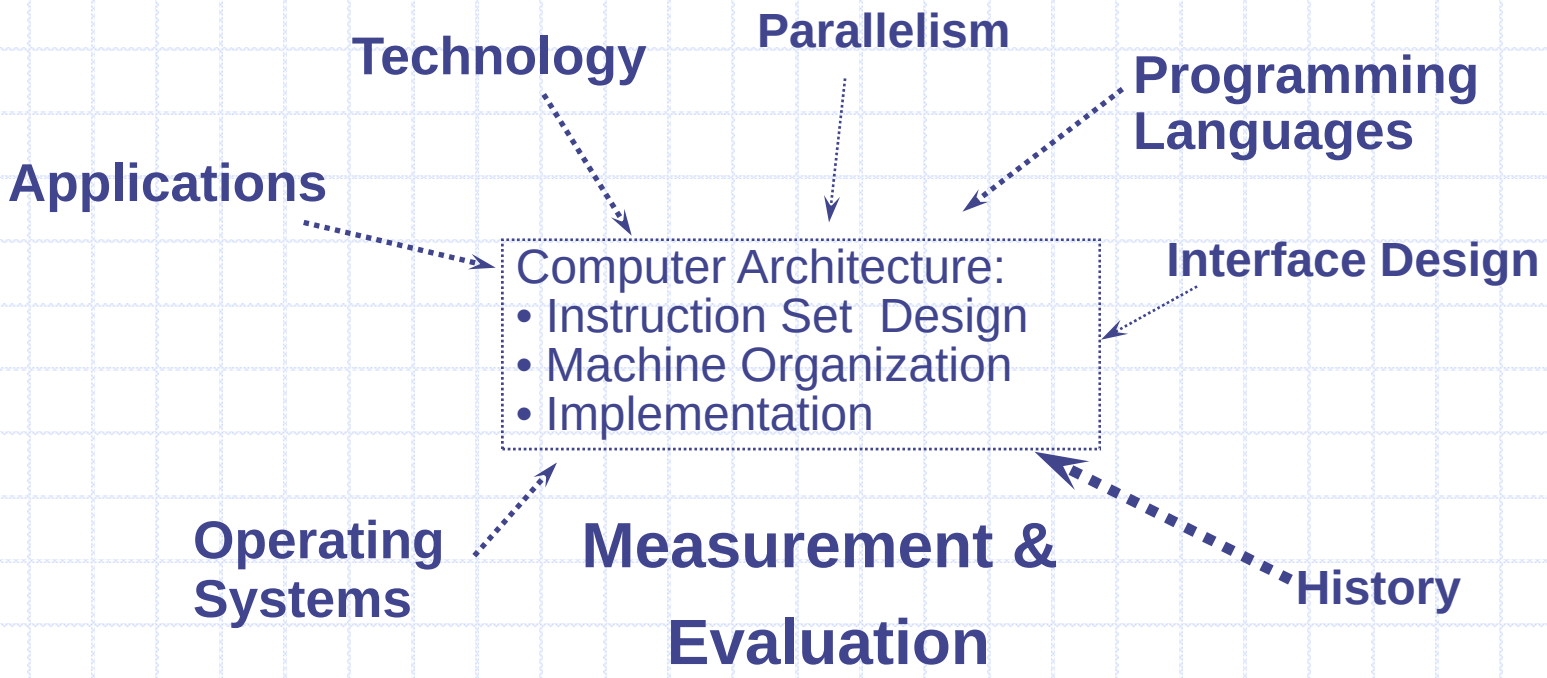
هر دوی سخت افزار و نرم افزار مرکب از لایه های سلسله مراتبی هستند، با هر لایه سطح پائینتر جزئیاتی از دید سطح بالاتر مخفی می شوند. این اصل تجرید، روشی است که طراحان سخت افزار و طراحان نرم افزار از عهده پیچیدگی سیستم های کامپیوتری بر آمدند.

یک محیط کلیدی بین لایه های انتزاعی معماری مجموعه دستورالعمل است: **فضای بین سخت افزار و نرم افزار سطح پائین** . این محیط مجازی توان بسیاری در پیاده سازی برای دگرگونی هزینه و کارایی در اجرای یک نرم افزار یکسان است.

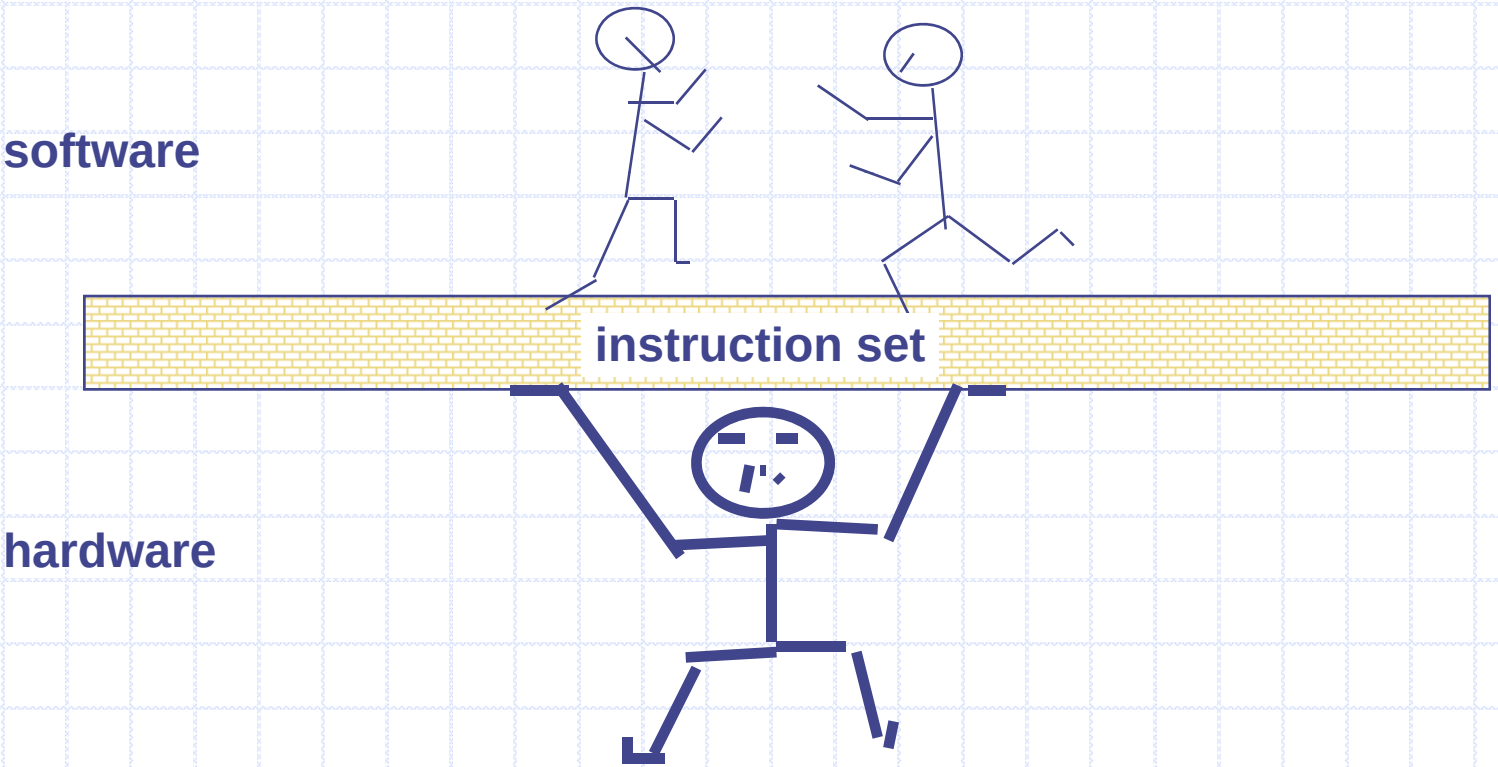
John L. Hennessy

David A. Patterson

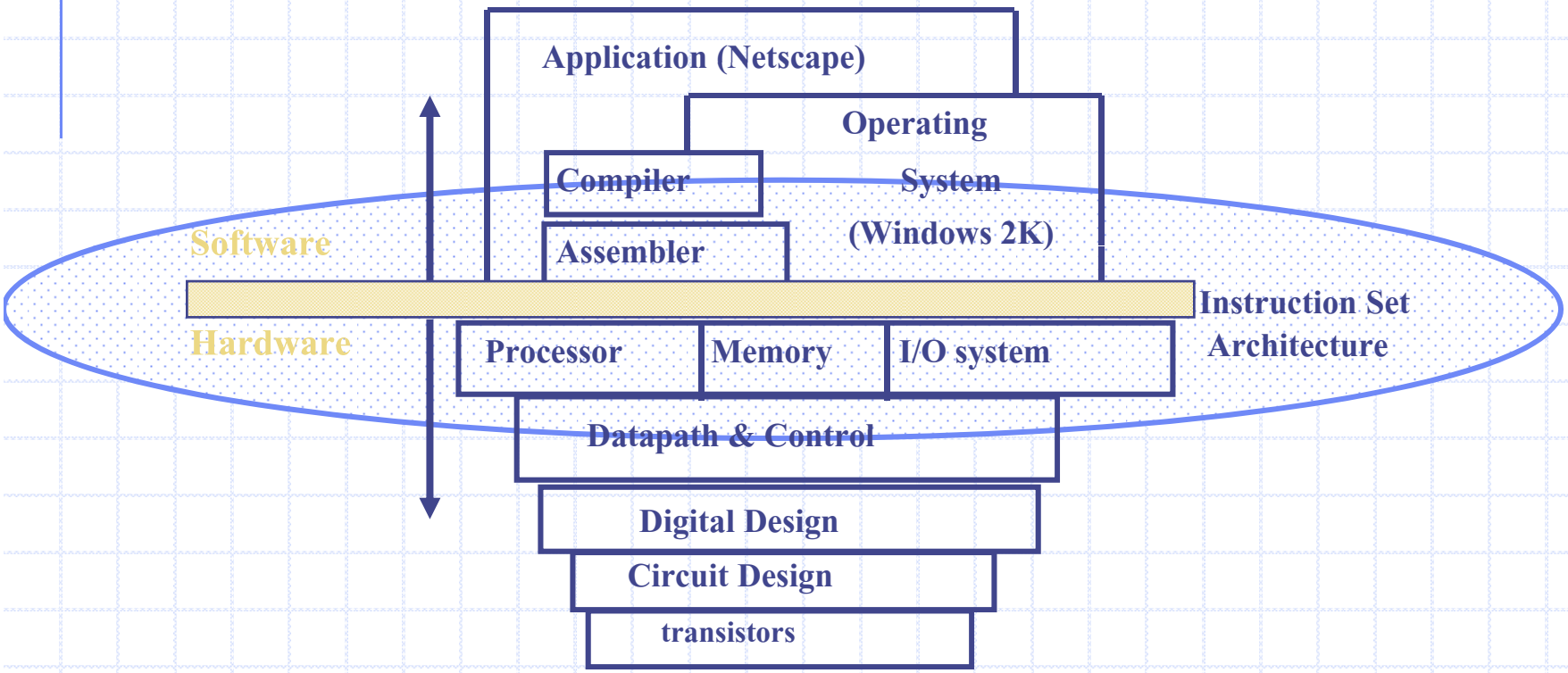
عوامل در معماری کامپیوتر



مجموعه دستورالعملها یک محیط بحرانی



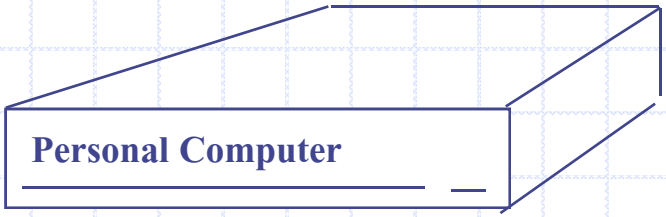
مهندسی و معماری کامپیوتر کجاست؟

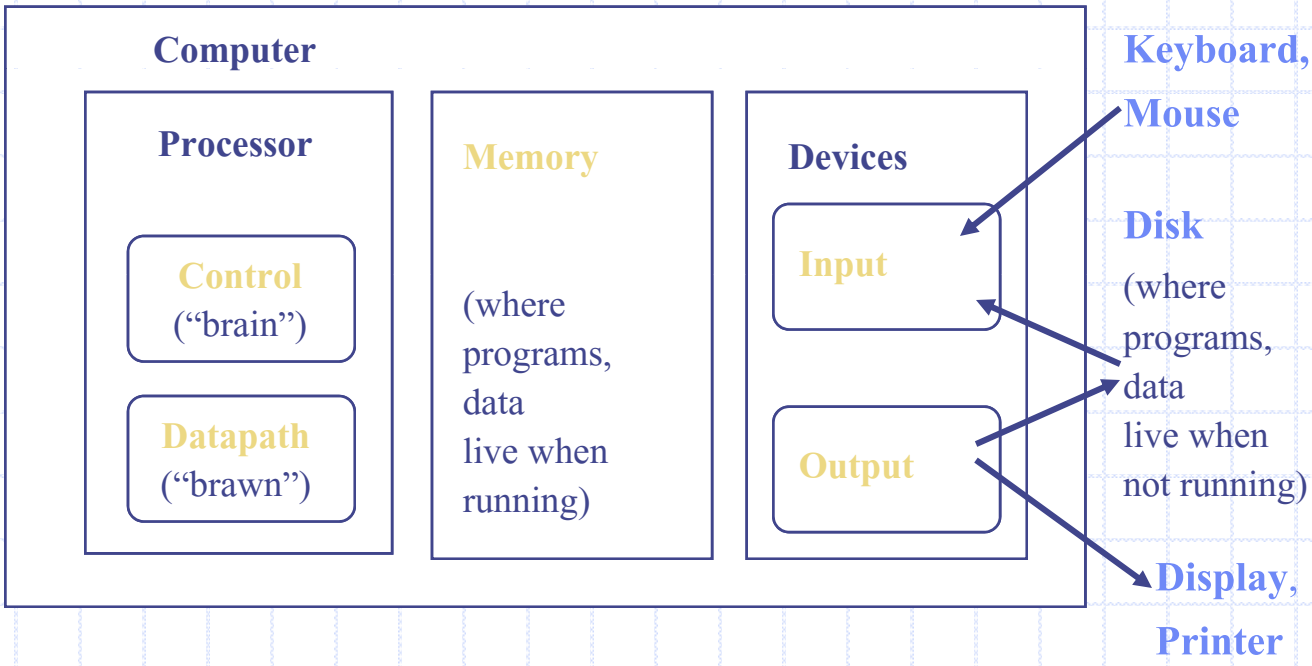


152

*هماهنگی بسیاری از لایه های انتزاعی

تشریح: پنج جزء ترکیب شده کامپیوتر

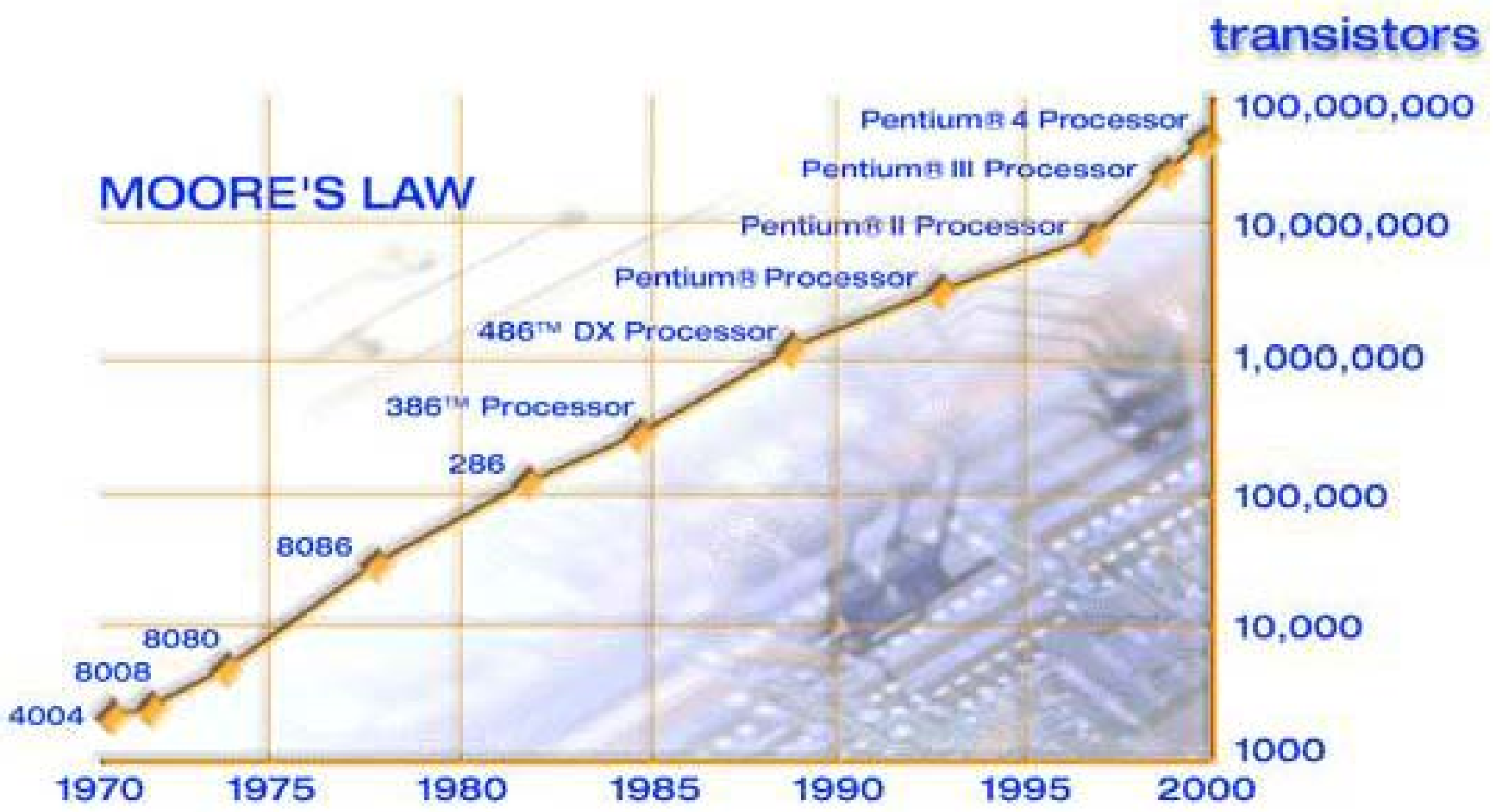




تکنولوژی کامپیوتر: تغییری مهیج

- پردازنده
  - هر یکسال و نیم، 2 برابر در سرعت(از سال 1985).
    کارایی 100 برابر، در دهه گذشته
- حافظه
  - ظرفیت DRAM: 2برابر در هر دو سال(از سال 96)
    بهبود اندازه 64x در دهه گذشته
- دیسک
  - ظرفیت: 2برابر در هر سال(از سال 97)
  - بهبود اندازه 250x در دهه گذشته

گرایش تکنولوژی: پیچیدگی ریزپردازنده

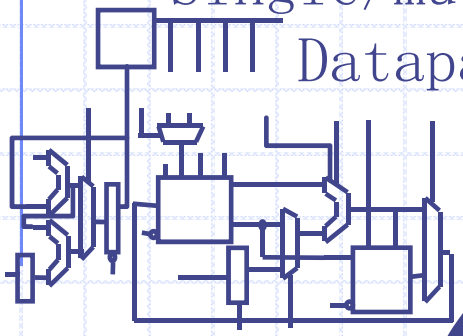


2X transistors/Chip Every 1.5 to 2.0 years

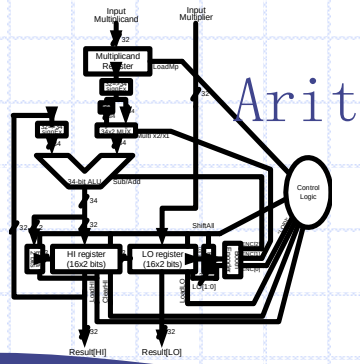
2برابر ترانزیستور/تراشه هر 1.5 تا 2 سال "موسوم به قانون مور"

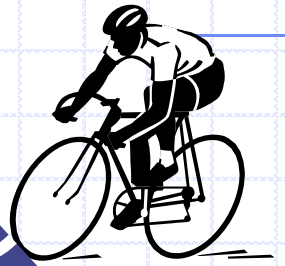
به کجا می رویم؟

Single/multicycle Datapaths

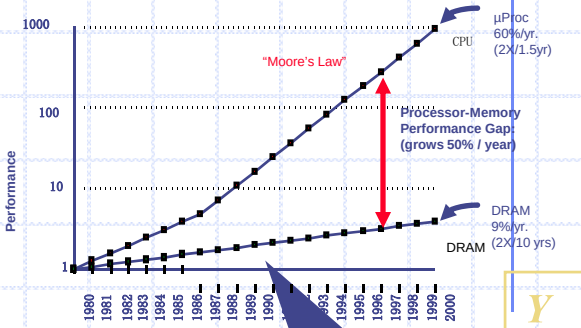


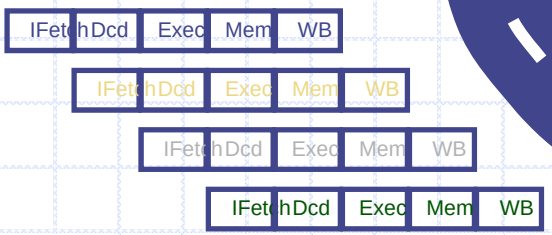
Arithmetic

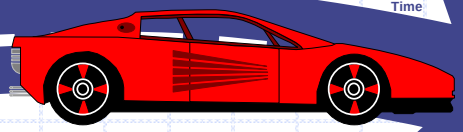




This course

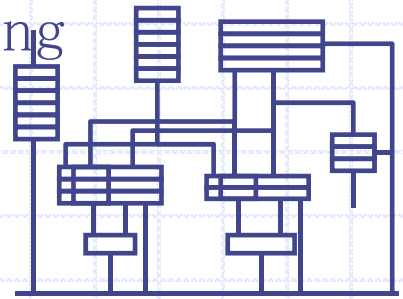


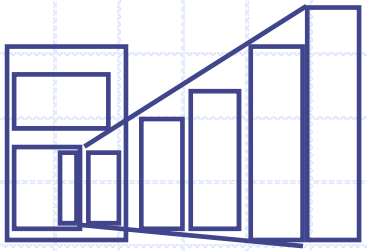




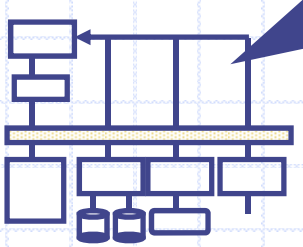
YOUR CPU

Pipelining





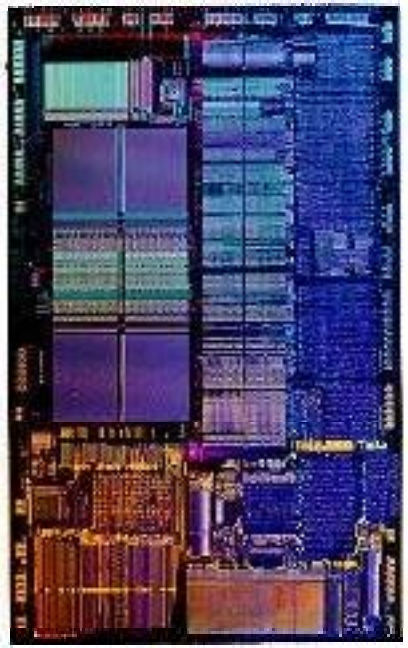
Memory Systems



I/O

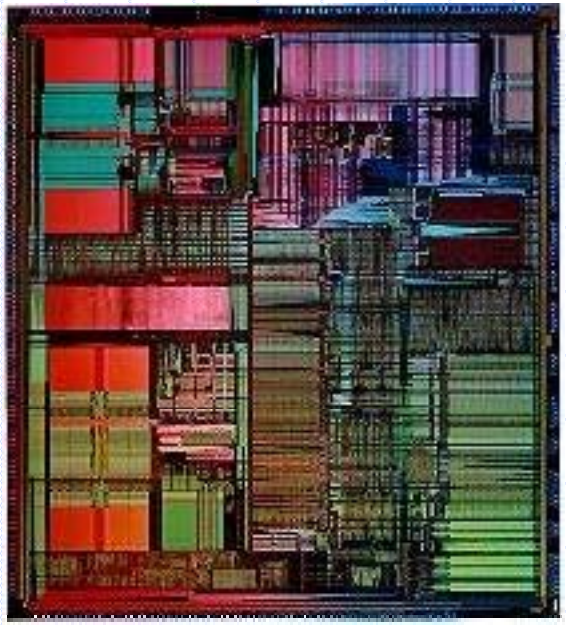
Intel 486™ DX CPU

- Design 1986 – 1989
- 25 MHz, 33 MHz
- 1.2 M transistors
- 1.0 micron
- 5 stage pipeline
- Unified 8 KByte code/data cache (write-through)
- First IA-32 processor capable of executing 1 instruction per clock cycle



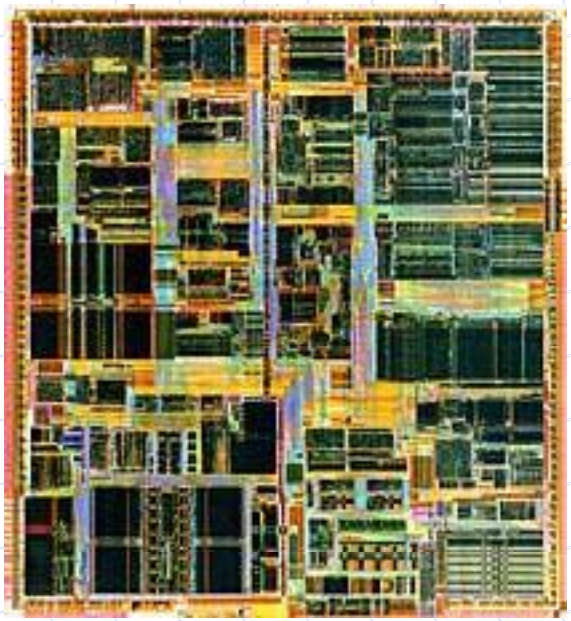
Pentium® Processor

- Design 1989 – 1993
- 60 MHz, 66 MHz
- 3.1 M transistors
- 0.8 micron
- 5 stage pipeline
- 8 KByte instruction and 8 KByte data caches (writeback)
- Branch predictor
- Pipelined floating point
- First superscalar IA-32: capable of executing 2 instructions per clock



Pentium® II Processor

- Design 1995 – 1997
- 233 MHz, 266 MHz, 300 MHz
- 7.5 M transistors
- 0.35 micron
- 16 KByte L1I, 16 KByte L1D, 512 KByte off-die L2
- First compaction of P6 microarchitecture



Pentium® III Processor (Katmai)

- Introduced: 1999
- 450 MHz, 500 MHz, 533 MHz, 600MHz
- 9.5 M transistors
- 0.25 micron
- 16 KByte L1I, 16 KByte L1D, 512 KByte off-chip L2
- Addition of SSE instructions.

SSE: Intel Streaming SIMD Extensions to the x86 ISA

Pentium® III Processor (Coppermine)

- Introduced: 1999
- 500MHz ... 1133MHz
- 28 M transistors
- 0.18 micron
- 16 KByte L1I, 16 KByte L1D, 256KByte on-chip L2
- Integrate L2 cache on chip, It topped out at 1GHz.

Pentium® IV Processor

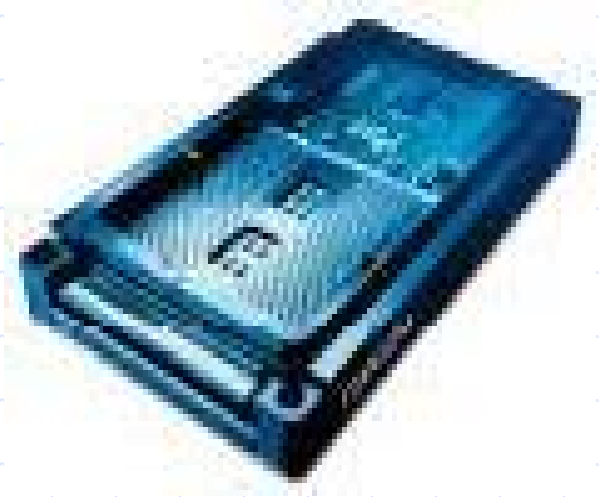
- Introduced: 2000
- 1.3GHz ... 2GHz ... 3.4GHz
- 42M ... 55M ... 125 M transistors
- 0.18 ... 0.13 ... 0.09 micron
- Latest one: 16 KByte L1I, 16 KByte L1D, 1M on-chip L2
- Very high clock speed and SSE performance

Intel® Itanium® Processor

- Design 1993 – 2000
- 733 MHz, 800 MHz
- 25 M transistors
- 0.18 micron
- 3 levels of cache
  - 16 KByte L1I, 16 KByte L1D
  - 96 KByte L2
  - 4 MByte off-die L3
- Superscalar degree 6, in-order machine
- First implementation of 64-bit Itanium architecture

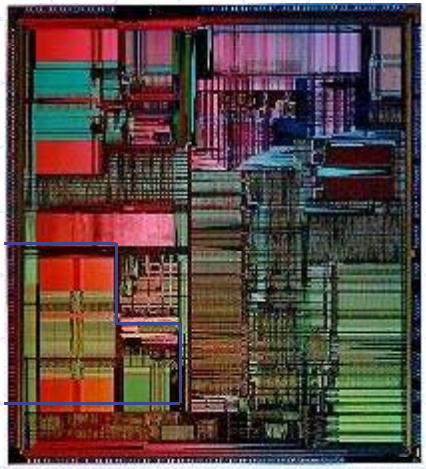
Intel® Itanium 2® Processor

- Introduced: 2002
- 1GHz
- 221 M transistors
- 0.18 micron
- 3 levels of cache
  - 32 KByte I&D L1
  - 256 KByte L2
  - integrated 1.5MByte L3
- Based on EPIC architecture
- Enhanced Machine Check Architecture (MCA) with extensive Error Correcting Code (ECC)



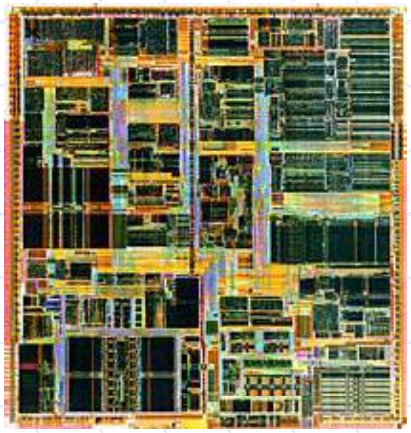
Cache Size Becoming Larger and Larger

1993: Pentium



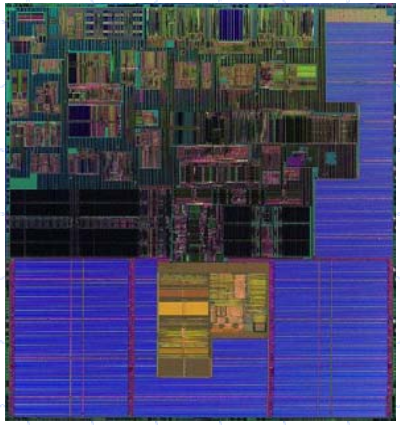
- 8 KByte I-cache and 8 KByte D-cache

1997: Pentium-II

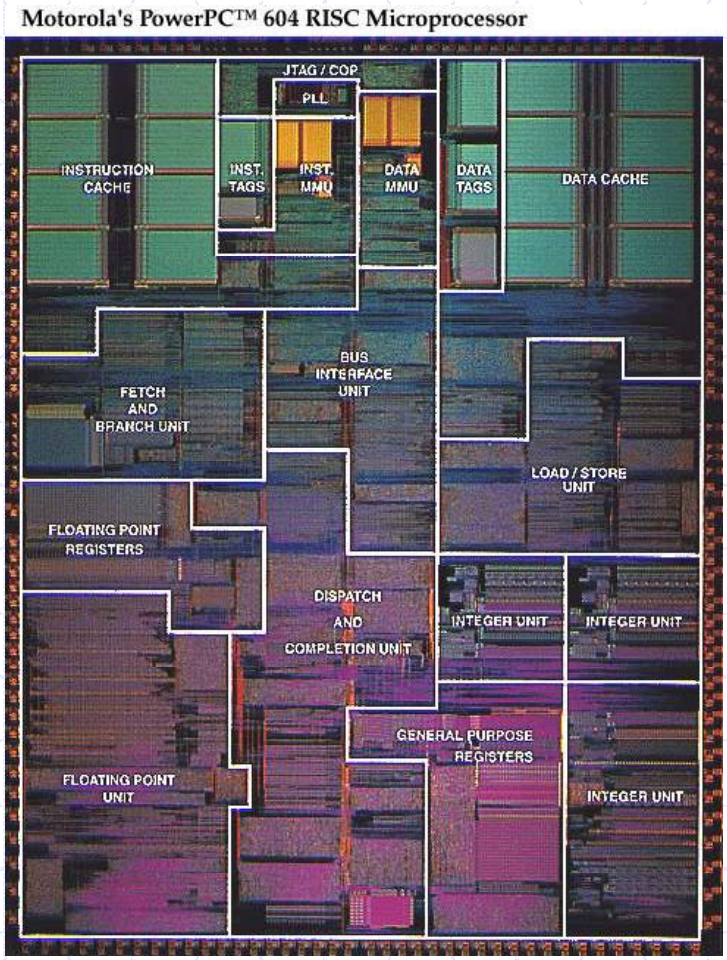


- 16 KByte L1I, 16 KByte L1D
- 512 KByte off-die L2

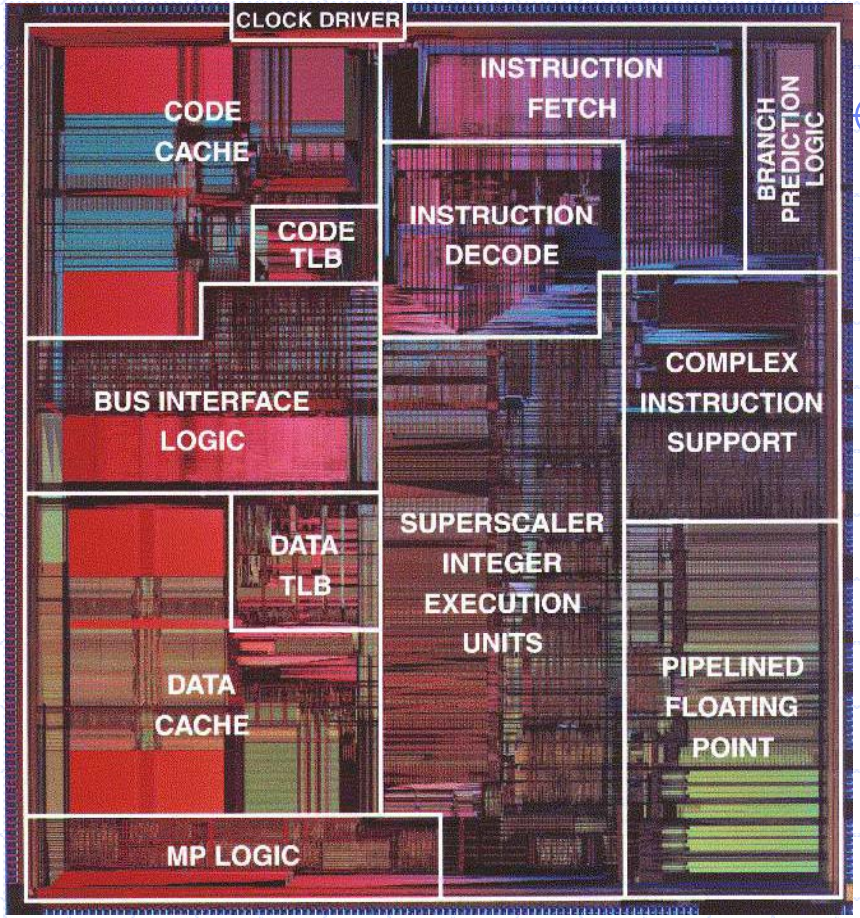
2002: Itanium-2



- Level 1: 16K KByte I-cache, 16 KByte D-cache
- Level 2: 256 KB
- Level 3: integrated 3 MB or 1.5 MB



Motorola's PowerPC 604



Pentium

Intel Pentium 4 Northwood

Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.

uOp Schedulers

FP Move Scheduler: (8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

Load / Store uOp Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Integer Execution Core

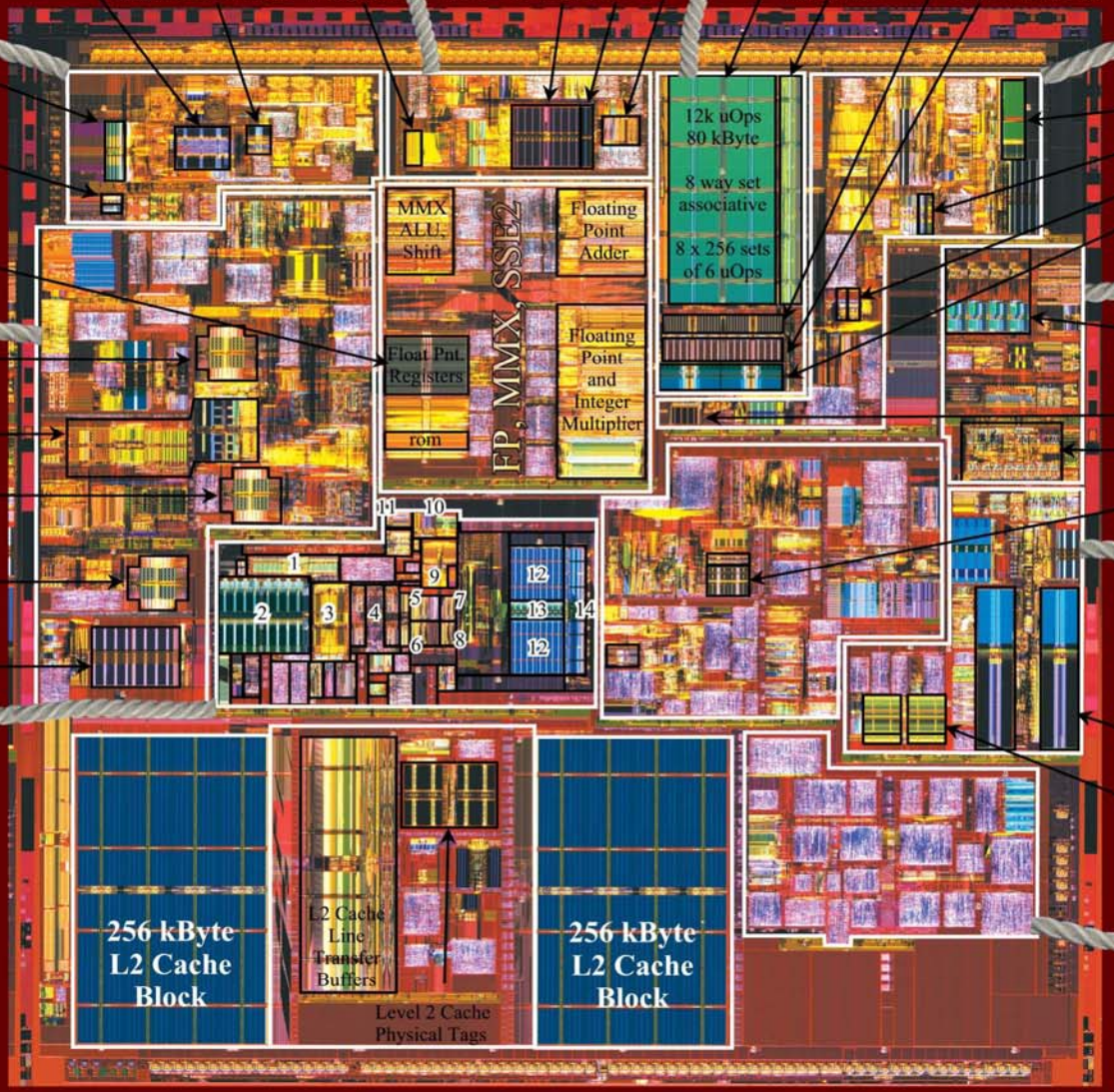
(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
(2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
(3) Databus switch & Bypasses to and from the Integer Register File.
(4) Flags, Write Back
(5) Double Pumped ALU 0
(6) Double Pumped ALU 1
(7) Load Address Generator Unit
(8) Store Address Generator Unit
(9) Load Buffer (48 entries)
(10) Store Buffer (24 entries)

Execution Pipeline Start

Register Alias History Tables (2x126)
Register Alias Tables
uOp Queue

Instruction Trace Cache

Micro code Sequencer
Micro code ROM & Flash
Trace Cache Fill Buffers
Distributed Tag comparators 24 bit virtual Tags



Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.
Return Stacks (2x16 entries)
Trace Cache next IP's (2x)
Miscellaneous Tag Data

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer
Trace Cache LRU bits
Raw Instruction Bytes in
Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total
Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

Front Side Bus Interface, 400..800 MHz

(11) ROB Reorder Buffer 3x42 entries
(12) 8 kByte Level 1 Data cache four way set associative. 1R/1W
(13) Summed Address Index decode and Way Predict
(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

April 19, 2003 www.chip-architect.com

Intel Pentium 5 Prescott

Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 1024 entries.

Return Stacks (4 x16 entries)

Trace Cache next IP's (4x)

Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle) Instructions with more than four are handled by Micro Sequencer

Raw Instruction Bytes in

Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total

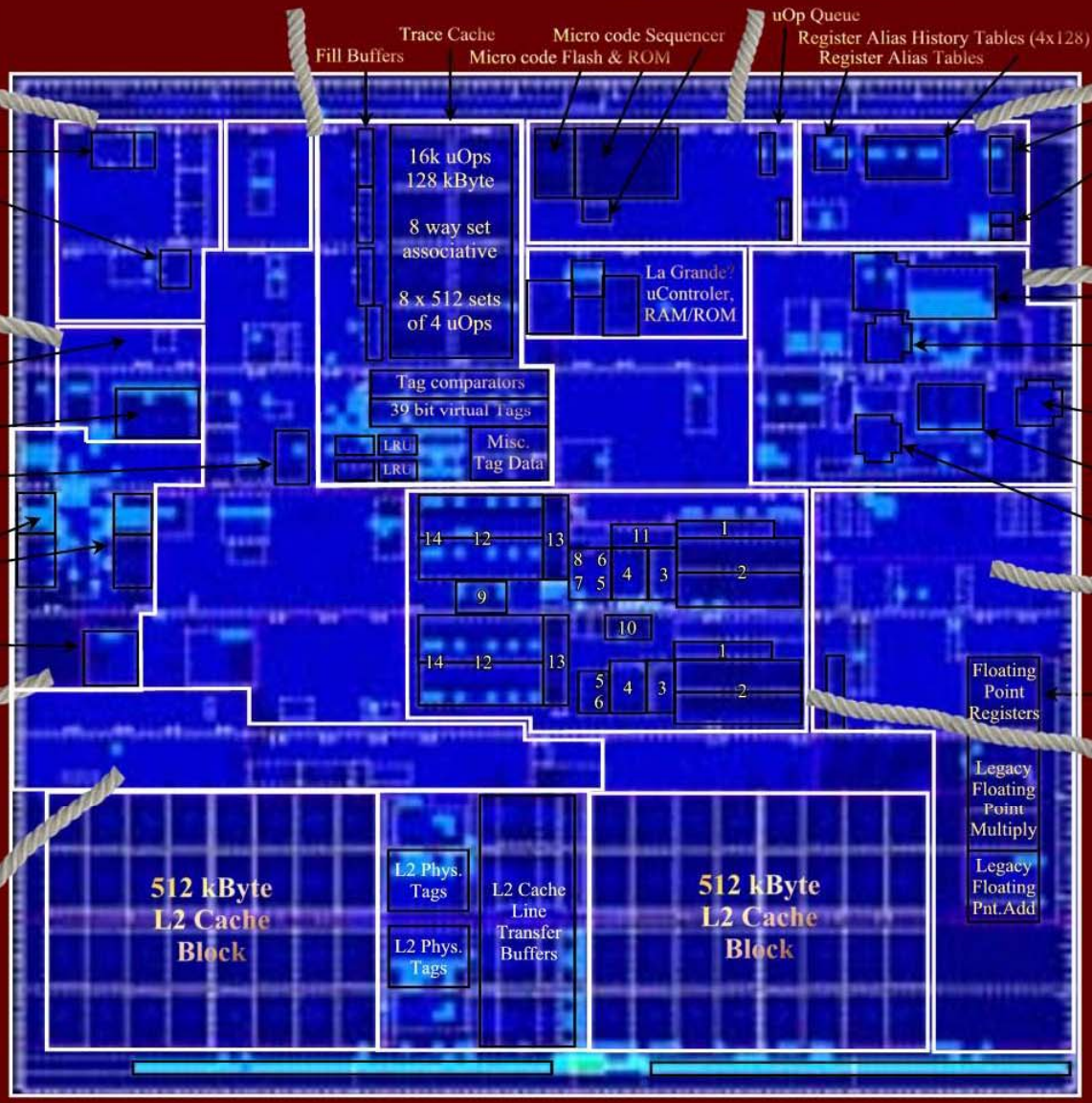
Instruction TLB's 128 entry, fully associative for 4k and 4M pages. In: Virtual address [47:12] Out: Physical address [39:12] + 2 page level bits

Instruction Fetch from L2 cache and Branch Prediction

Front Side Bus Interface, 533..800 MHz

Instruction Trace Cache

Execution Pipeline Start



Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

uOp Schedulers

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

FP Move Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

Load / Store uOp Scheduler: (8x8 dependency matrix)

FP, MMX, SSE1..3

Floating Point, MMX, SSE1..3 Renamed Register File 256 entries of 128 bit.

Integer Execution Core

(1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle

(2) Integer Renamed Register File 256 entries of 32 bit (+ 6 status flags) 12 read ports and six write ports

(3) Databus switch & Bypasses to and from the Integer Register File.

(4) Flags, Write Back

(5) Double Pumped ALU 0

(6) Double Pumped ALU 1

(7) Load Address Generator Unit

(8) Store Address Generator Unit

(9) Load Buffer (96 entries)

(10) Store Buffer (48 entries)

(13) Databus multiplexing

(14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

(11) ROB Reorder Buffer 4x64 entries

(12) 16 kByte Level 1 Data cache four way set associative. 1R/1W

April 19, 2003 www.chip-architect.com

یک بازبینی از جریان طراحی پردازنده

چگونه یک CPU طراحی می شود

- طراحی معماری مجموعه دستورالعمل (ISA design)
- طراحی در سطح وظایف RTL design (function-level)
- طراحی در سطح اجزاء ترکیب دهنده
- طراحی gate-level/switch-level
- طراحی در سطح مدار

روش کلاسیک طراحی مرحله ای معماری مجموعه دستورالعمل

- انتخاب یک الگوی ساخت A
- تعریف A برای تطبیق با:
  - کارایی مورد تقاضای جدید و تکنولوژی جدید
- ارزیابی (شبیه سازی معماری مجموعه دستورالعمل)
- تکرار تا کسب رضایت

کل استراتژی شبیه سازی

1. **شبیه ساز در سطح دستورالعمل (ISA):** این روش برای ارزیابی کارایی در سطح مجموعه دستورالعمل برای شرح بیشتر مدل سازی استفاده می شود.

2. **شبیه سازی در سطح سیستم:** مدلهای این شبیه ساز جزئیات مربوط به محیط سیستم شامل برخی چیزها مانند وقفه ها و مدیریت حافظه را مدل سازی می کند.

کل استراتژی شبیه سازی

(Con'd)

3. **در سطح RTL:** مدلهای این شبیه ساز تشریح RTL از طراحی است.

4. **در سطح سوئیچ همراه با تاخیرها:** بیش از همه برای شبیه سازی مولفه های طراحی به کار می رود؛ بردارهای آزمایشی از مرحله RTL تولید شده اند.

5. **شبیه سازی در سطح مدار:** برای مدل سازی جزئیات مسیر بحرانی به علاوه برای بازبینی مدارها در تغییرات دمایی، توان ارائه شده و غیره به کار می رود.

کارایی شبیه سازها

Simulator	Level of Accuracy	Simulation Rate
ISA	Instruction set	> 10^6 cycles/second
System	System level (OS instructions + interrupts)	> 10^3 cycles/second
RTL	Synchronous register transfer	> 10 cycles/second
Gate	Gate/switch level	> 1 cycles/second

تعداد چرخه های شبیه سازی شده در ثانیه بر روی یک کامپیوتر میزبان

کارایی دستورالعمل بر چرخه (IPC)

- ایجاد مدل کارایی که:
  - قابل انعطاف
  - پذیرای پارامتر(Parameterized) از طریق دستگیره
  - دارای دقت ساعت در مقایسه با RTL برابر با 95% باشد
  - به صورت قابل توجهی سریعتر از RTL
- مدلها مرکب از دو بخش هستند.
  - شبیه ساز مجموعه دستورالعمل -> اجرا کننده محک(benchmark)
  - شبیه ساز خط لوله -> "حسابدار" برای چرخه های ساعت
- سرعت های شبیه سازی
- اجرای محک ها(benchmark)، به روز رسانی ریز معماری بر طبق آن
- چرخهٔ : کد -> شبیه ساز -> characterize -> وفق دادن(tune)

فصل دوم
معماری مجموعه دستورالعملها

- مقدمه
- یک بررسی موردی: معماری مجموعه دستورالعمل ماشین MIPS

مراحل اجرای یک دستورالعمل

واکشی دستورالعمل: برداشت دستورالعمل بعدی از حافظه

کدبرداری از دستورالعمل: بررسی دستورالعمل برای مشخص شدن اینکه:

چه عملی باید توسط دستورالعمل انجام گیرد (به عنوان مثال جمع)

چه عملوندهایی مورد نیازند، و نتایج باید کجا قرار گیرند.

واکشی عملوندها: عملوندها برداشت می شوند.

اجرا: اجرای عملیات بر روی عملوندها

بازنویسی نتیجه: نوشتن نتیجه در محل مخصوص

دستورالعمل بعدی: تعیین اینکه دستورالعمل بعدی از کجا گرفته شود.

چه چیزی در یک ISA(معماری مجموعه دستورالعمل) مشخص می شود؟

کدبرداری از دستورالعمل: اعمال و عملوندها چگونه تعیین می گردند؟

واکشی عملوندها: عملوندها ممکن است کجا باشند؟ چه تعداد؟

اجرا: چه اعمالی می تواند انجام گیرد؟ چه نوع داده و چه اندازه هایی؟

بازنویسی نتایج: نتایج کجا نوشته می شوند؟ چه تعداد؟

دستورالعمل بعدی: دستورالعمل بعدی را چگونه می توان انتخاب نمود؟

یک ISA ساده:حافظه به حافظه

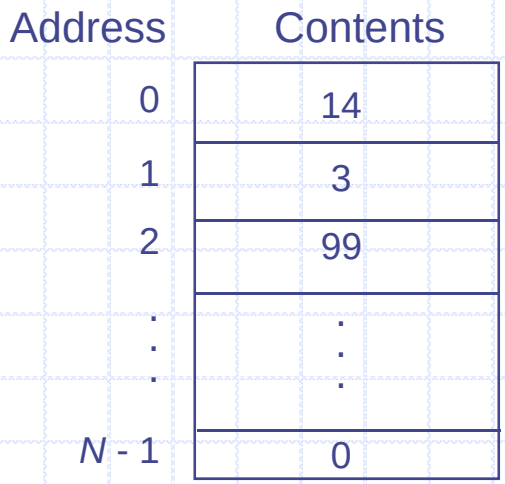
- چه عملیاتی می تواند اجرا شود؟ عملیات پایه ریاضی (برای این لحظه)
- چه نوع داده و چه اندازه ای؟ نوع داده صحیح 32 بیتی (integer (32
- عملوندها و نتایج کجا می توانند قرار گیرند؟ حافظه
- چه تعداد عملوند و نتیجه؟ 2 عملوند، 1 نتیجه
- اعمال و عملوندها چگونه مشخص می شوند؟

OP DEST,SRC1,SRC2 عمل مقصد,منبع1,منبع2

- چگونه می توانیم دستورالعمل بعدی را انتخاب کنیم؟ بعدی به ترتیب

مدل حافظه

حافظه را به عنوان یک آرایه بزرگ از n عدد صحیح در نظر بگیرید، که بوسیله اندیس قابل دستیابی است.(حافظه با دستیابی تصادفی موسوم به ram)



به عنوان نمونه ، M[1] شامل مقدار 3 است. ما می توانیم در این مکانها نوشتن و خواندن را انجام دهیم. این مکانها صرفا در دسترس ماست. تمام مکانهای "مجرد" (از قبیل متغیرها در C) باید مکانهایی را در M تعیین کنند.

ترجمه کد ساده

کد C مفروض

```
A = B + C;
```

ما می توانیم تصمیم بگیریم که متغیر A مکان 100 ، B مکان 48 و C مکان 76 را

اشغال می کند. کد بالا را به معادل کد اسمبلی آن تبدیل می کنیم:

```
ADD  M[100],  M[48],  M[76]
```

چگونه می توانیم عبارت زیر را تبدیل کنیم

```
A = (B + C) * (D + E);
```

استفاده از یک مکان موقتی

فرض کنید ما A را در 100، B را در 48 ، C را در 76، D را در 20 و E را در32 قرار می دهیم.

اکنون یک مکان بدون استفاده را انتخاب می کنیم (مثلا 84)

```
# A = B + C          ADD  M[100],  M[48],   M[76]
# temp = D + E       ADD  M[84],   M[20],   M[32]
# A = A * temp       MUL  M[100],  M[100],  M[84]
```

مشکلات در معماری حافظه به حافظه

- حافظه اصلی خیلی کندتر از مدارات محاسباتی است

  - این مطلب از سال 1950 تا 2003 درست است!

- خانه های زیادی برای مشخص نمودن آدرسهای حافظه گرفته می شوند.
- معمولا نتایج یک یا دو دستورالعمل بعد مورد استفاده قرار می گیرند.

به خاطر داشته باشید: بخشهای اشتراکی را سریعتر نمائید!

راه حل: نتایج میانی یا موقتی را در حافظه های سریع و نزدیک به واحد محاسبه ذخیره نمائید.

ماشینهای مبتنی بر انباشتگر(Accumulator)

یک ماشین انباشتگر، یک بافر پرسرعت واحد (مانند یک مجموعه از D latchها یا فلیپ فلاپها ، هر کدام برای یک بیت داده) را نزدیک واحد محاسبه منطق نگهداری می کند.

در ساده ترین حالت، فقط یک عملوند می تواند مشخص گردد; انباشتگر به صورت مجازی به مفهوم "OP Operand" می باشد یعنی:

acc. = acc. OP operand

Example:

# Load B into acc.	LOAD M[48]
# Add C to acc. (now has B+C)	ADD M[76]
# Write acc. To A	STORE M[100]

ماشین مبتنی بر انباشتگر که A=(B+C)*(D+E) را انجام می دهد

Load D into acc. LOAD M[20]

بارگذاری Dدر انباشتگر

Add E to acc. (now has D+E) ADD M[32]

جمع E با انباشتگر.(اکنون داریم D+E)

Write acc. To A STORE M[100]

نوشتن محتوای انباشتگر در A.

Load B into acc. LOAD M[48]

بارگذاری Bدر انباشتگر.

Add C to acc. (now has B+C) ADD M[76]

جمع نمودن Cبا انباشتگر.(اکنون داریم B+C)

Multiply A to acc. MUL M[100]

ضرب کردن A در انباشتگر.

Write (B+C) * (D+E) to A STORE M[100]

نوشتن نتیجه در A

ضعف ماشینهای مبتنی بر انباشتگر

- هنوز نیازمند ذخیره سازی بسیاری مقادیر موقتی و میانی در حافظه می باشیم
- در واقع انباشتگر فقط برای یک ترتیب از محاسبات که در آن نتیجه یکی، ورودی برای بعدی است، مفید می باشد.

ماشینهای مبتنی بر انباشتگر هنوز در کامپیوترهای اولیه معمول بودند

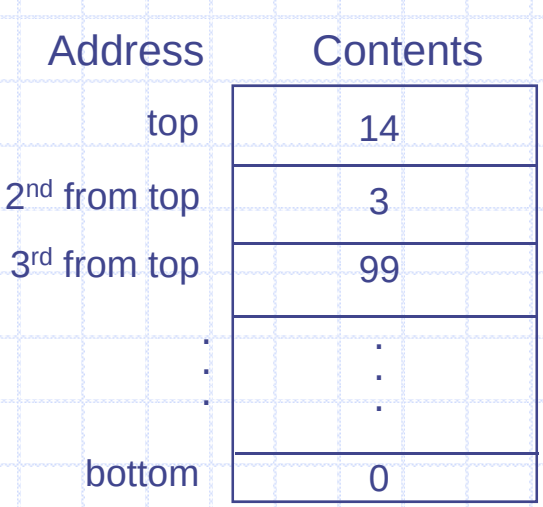
- یک طراحی ساده، و بنابراین محبوب ، مخصوص برای
  - کامپیوترهای اولیه
  - میکروپروسسرهای اولیه (4004 ، 8008)
  - مدلهای Low-end (ارزان)

پیشنهادات برای ماشینهای مبتنی بر انباشتگر

- اگر منابع سخت افزاری بیشتری در دسترس است، مکانهای ذخیره سازی سریع را در کنار انباشتگر قرار دهید:
- ماشینهای مبتنی بر پشته
- ماشینهای مبتنی بر ثبات
  - خاص منظوره
  - عام منظوره

ماشینهای مبتنی بر پشته

ایده: یک ستون از مکانهای ذخیره سازی سریع دارای یک بالا(top) و یک پایین (bottom)



یک دستورالعمل فقط از مقدار بالای پشته(top) قابلیت برداشت دارد، یا شاید دو یا سه مقدار به عنوان بالای پشته در نظر گرفته شوند.

ما می توانیم مقادیر جدید را در بالای پشته قرار دهیم ("push") یا از بالای پشته برداریم ("pop") اما فقط همین. ما نمی توانیم به مکانهایی در زیر پشته دسترسی داشته باشیم مگر اینکه هر چیز بالای آن را خارج کنیم.

معماری مجموعه دستورالعمل ماشینهای مبتنی بر پشته

اعمال اصلی شامل:

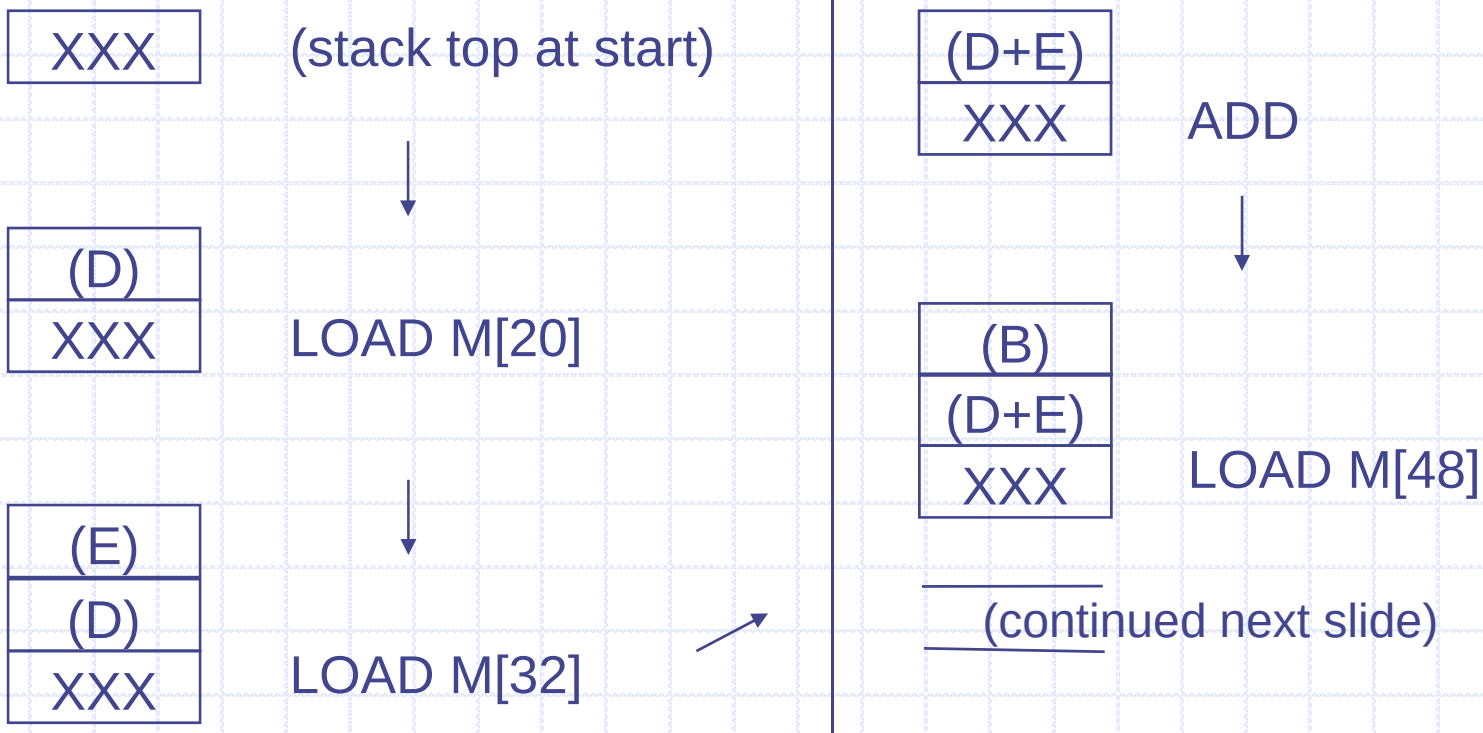
بارگذاری: برداشت مقدار از حافظه و قرار دادن آن بر روی پشته

ذخیره سازی: برداشت مقدار از پشته و ذخیره آن در حافظه

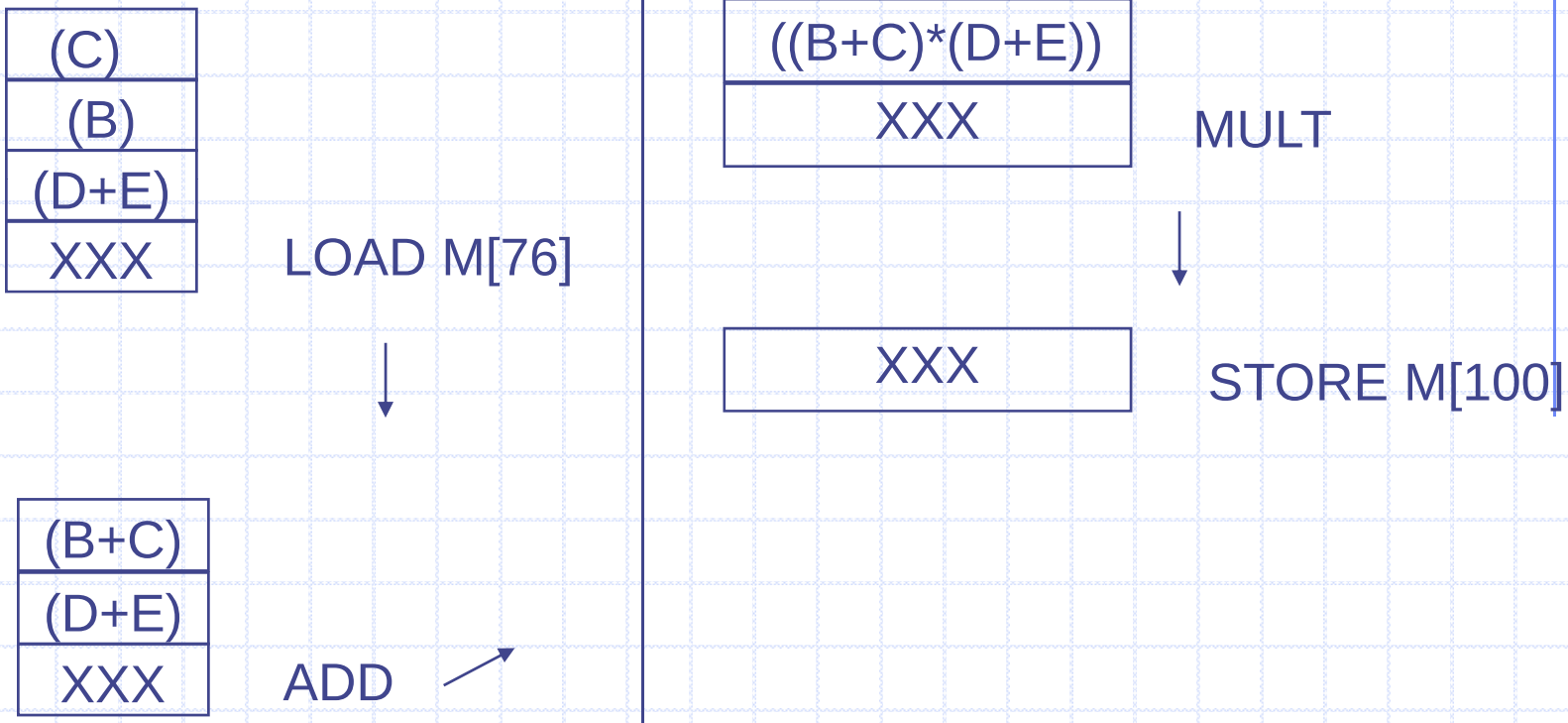
حسابی: خارج کردن یک یا دو مقدار از پشته؛ قرار دادن نتیجه روی پشته

دونسخه ای (Dup): گرفتن مقدار از بالای پشته بدون حذف آن؛ قرار دادن یک کپی جدید از آن در روی پشته (چرا این عمل کاربردی است؟)

ماشین مبتنی بر پشته که A=(B+C)*(D+E) را انجام می دهد



ماشین مبتنی بر پشته



دقت کنید که اکنون پشته مشابه زمانی است که شروع نمودیم.

کاربرد ماشینهای مبتنی برپشته

- بسیاری از کامپیوترهای اولیه
- واحد ممیز شناور در 8086 (تقریبا)
- ماشین مجازی جاوا

ماشینهای مبتنی بر ثبات

ایده:به کار بردن مکانهای ذخیره سازی زیاد("ثباتها")نزدیک انباشتگر

- ثبات ها نام / شماره های مشخصی دارند که می توانند به جای حافظه استفاده می شوند
- دستیابی نسبت به حافظه اصلی
- خیلی سریعتر است

(1-2 CPU cycles vs. ~ 100 CPU cycles on PC)_

- ثبات ها نسبت به مکان های حافظه خیلی نزدیک ترند

_MIPS 32 تا ثبات 32 بیتی دارد

_ثبات های کمتر آدرس های کوچکتر وتعداد بیتهای کمتر برای نام گذاری آنها

_منابع استفاده کمتر است و باید با دقت از آنها استفاده شود!

ثباتهای عام منظوره_خاص منظوره

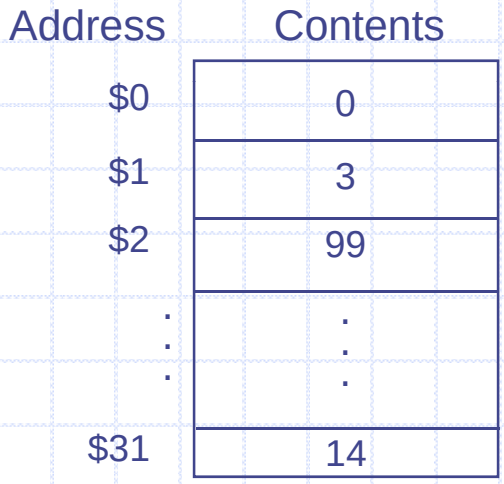
- یک ثبات خاص_ منظوره برای اهداف مشخص استفاده می شود و ممکن است عملیاتی را که استفاده می شود را محدودکند
  - طراحی آسانتر سخت افزار: ثبات را در جایی که دقیقا نیازاست قرار بده
  - به منظور استفاده موثر برای کامپایلر خیلی سخت تر است.
- یک ثبات عام منظوره می تواند دربیشتر مسیرهای عملیاتی استفاده شود بنابراین مسیریابی خیلی مشکل است

ثبات های خاص منظورهThe z_80 cpu

- هفت تا ثبات 8 بیتی: (می توانند جفت شوند A,B,C,D,E,H,LBC,DE,HL).
- سه تا ثبات 16 بیتی: PC 62(شمارنده برنامه).
- جمع ,تفریق, شیفت تنها با A می تواند انجام شود (شمارنده 8 بیتی).
- افزایش و کاهش می تواند با تمام ثبات ها وجفتهای ثبات انجام شود .
- می توانند از حافظه آدرس (HL)را واکشی کنند و در هر8بیت ثبات قرار دهند.
- یک واکشی از آدرس(BC)یا(DE)تنها می تواند به Aبرود.
- واکشی ها از (HL),(BC)و (IX)تعداد چرخه های متفاوتی می گیرد
- چه کسی می خواهد برای این یک کامپایلر بنویسد؟

ثبات عام منظوره ماشین های (GPR)

MIPS(وپردازشگرهای مشابه)32تا ثبات عمومی دارند (GPRs)هر32بیتlongهستند .همه می توانند نوشته یا خوانده شوند به جز ثبات صفر که همیشه صفراست ونمی تواند تغییر کند.زمان دستیابی به ثبات ثابت است.



ماشین GPR A=(B+C)*(D+E) را انجام می دهد

#R1 = B + C	M[48], M[76]	$1	ADD
#R2 = D + E	M[20], M[32]	$2	ADD
#A = R1 * R2	M[100],	$1, $2	MUL

اندازه های داده های متفاوت

چطورباید با اندازه های داده های متفاوت رفتارکنیم؟
انتخاب یک اندازه برای یک واحد ذخیره شده در یک آدرس تنها

- ذخیره کردن داده بزرگ در یک مجموعه از مکانهای همجوارحافظه
- ذخیره کردن داده کوچک در یک مکان:

use shift & mask ops

امروزه تقریبا همه ماشینها(شاملMIPS)دارای آدرس دهی بایتی "Byte_Adressable"هستند هر مکان آدرس دهی ، در حافظه 8بیتی نگهداری می شود.

حافظه MIPS

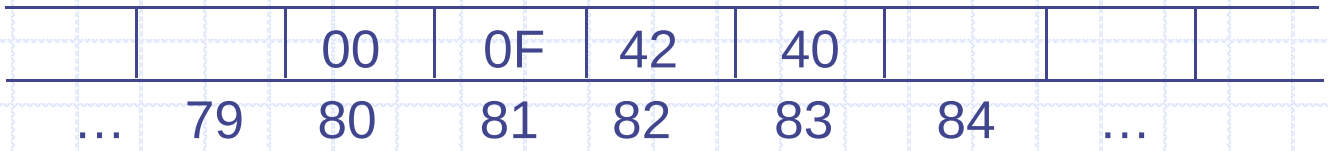
روی یک ماشین قابل آدرس دهی بایتی ازقبیلMIPS اگرما بگوییم یک کلمه (32بیت) درآدرس80 ذخیره شده به این معناست که مکانهای 80تا83را اشغال کرده.(کلمه بعدی از84 شروع می شود).

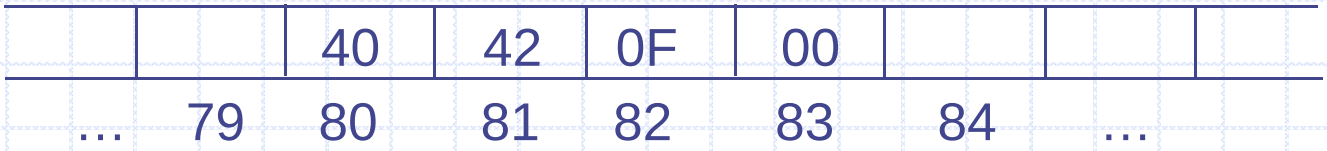
به طور نرمال بارگذاری وذخیره چندین بیت باید "تنظیم" شود.آدرس nبایتی بارگذاری یا ذخیره باید مضربی ازnباشد.برای نمونه نیم کلمه تنها در آدرسهای زوج ذخیره میشود.

MIPS اجازه بارگذاری شدن و ذخیره شدن برای استفاده مخصوص دستورالعمل ها را نمی دهند اما آنها ممکن است کندترشوند(بیشتر پردازشگرها این را برای همیشه اجازه نمی دهند!).

Byte-Order (“Endianness”)

- برای یک داده چند بایتی کدام قسمت به کدام بایت می رود؟
- اگر$1محتوی($H1000000(F4240و ما آن را در آدرس 80 ذخیره کنیم :
- در یک ماشین “Big End” “Big Endian” به آدرس 80 می رود
- در یک ماشین “Little Endian” از سوی دیگر است.





Big-Endian vs. Little-Endian

- ماشینهای Big Endian:MIPS,SPARC,68000
- ماشینهای Little Endian:بیشتر پردازشگر های intel,Alpha,Vax
- سازگاری مشکلات انتقال چندین بایت داده بین ماشینهای Big Endian,Little Endian

روشهای آدرس دهی

یک روش آدرس دهی ISA's این سوال را جواب می دهد
:عملوندها کجا میتوانند ذخیره شوند؟

ما دو نوع ذخیره سازی در MIPSداریم(وبیشتر ماشینهای دیگر)
:ثباتها و حافظه اصلی.ما می توانیم به هریک از این دو یا هردو عملوندها برویم . یک تک عملوند می تواند با هر یک از این دو بیاید یک ثبات یا یک مکان حافظه ، و روشهای آدرس دهی راه های گوناگون تشخیص این مکانها را ارائه می کند.

روشهای آدرس دهی ساده

دراین روشها یک مکان یا داده به طور مستقیم دریک دستورالعمل داده می شود:

Mode name	Example	Meaning
Register	mov $1, $2	R2 R1
Direct (or absolute)	mov $1, (40)	M[40] R1
Immediate	mov $1, #40	40 R1

روشهای آدرس دهی غیر مستقیم

در تولید یک آدرس حافظه یکی یا بیشترثباتها استفاده می شوند

Mode name	Example	Meaning
Reg. Indirect	mov $1, ($2)	M[R2] R1
Displacement	mov $1, 40($2)	M[40+R2] R1
Indexed	mov $1, $4($2)	M[R4+R2] R1
Mem. Indirect	mov $1, @($2)	M[M[R2]] R1

روشهای آدرس دهی پیشرفته

اجزای زیادی اجزای اصلی را در زبانهای سطح بالا پشتیبانی می کنند یا تعداد دستور العمل هارا در طول دستیابی از حافظه افزایش می دهند

Mode name	Example	Meaning
Auto-increment	mov $1, 4($2) ++	M[4+R2] R1
Auto-decrement	mov $1, 4($2) - -	M{R2-4] R1
Scaled	mov $1, 40($2) [s]	M[40+R2x2s] R1

روشهای آدرس دهی منتخب

کارهایی که انجام می شود :هر روش آدرس دهی ممکن است برای هر عملوندی در هر زمان استفاده شود

- طرح دستورات سطح بالا به طور مستقیم به دستورالعملها آسانتر است.
- برای طراحی پردازشگر سخت است بعلت اینکه باعث پیچیدگی بیشتر می شود

آدرس دهی های محدود: تنها روشهای کمی را اجازه می دهند ویا تعداد عملوندها را به روش های معین محدود میکنند

- برای کامپایلر یا برنامه نویس که پیرو قواعد مشخص هستند سخت تراست
- شایدطول کد زیاد شود

بسامد روشهای آدرس دهی

سه برنامه روی(مینی کامپیوتر)vaxمشخص شده که همه انواع روشها را پشتیبانی می کنند:

Mode Name	Frequency of mode (%) Min.	ave.	max.
Displacement	32	42	55
Immediate	17	33	43
Reg. Indirect	3	13	24
Scaled	0	7	16
Mem. Indirect	1	3	6
Others	0	2	3

روشهای آدرس دهی داده های تجربی

- بزرگی تغییر مکان مورد نیاز چه طور باید باشد؟

در بررسی های Spec in92وSpec fp92 99درصد تغییر مکانها در دامنه 215-+انجام می شود.

- بزرگی ثابت فوری مورد نیاز چه طور باید باشد؟

در مطالعات نشان داده شده 50 تا 60 درصد در 8 بیت و75 تا 80 درصد در 16 بیت است.

چه طور ما نمایش دادن دستورالعمل ها را انجام می دهیم ؟

- ما به چند بیت احتیاج داریم برای اینکه بگوییم چه عملی انجام شده (مثلا جمع ,تفریق ,ضرب ,غیره)که به این کدگذاری عملوند opcodeگویند
- ما به چند بیت برای هر عملوندو نتیجه احتیاج داریم(در نمونه ما جمعا 3تا)
- چه نوع روش آدرس دهی
- شماری از ثباتها , آدرس حافظه و یا ثابت فوری

دستورالعملهای با طول متغیر

- تا کنون Vax هر روش را برای هر عملوندی اجازه داده می تواند یک دستور العمل با سه تا آدرس های 32 بیتی باشد (آدرس دهی مستقیم) 12 بایت در این دستورالعمل
- اما ثباتها به بیتهای کمی برای مشخص شدن نیاز دارند بنابراین 12 بایت برای یک دستورالعمل که تنها از 3 تا ثبات استفاده می کنند هدر میشود .
- باید از دستور العمل های با طول متغیر استفاده کرد.درVaxدستورالعملهل از 1 تا 17 بایت متغیراند.

دستورالعمل های با طول ثابت

- اگر هر دستورالعمل با تعداد بیتهای یکسان (ترجیحا یک عدد زوج مانند 16 یا 32)بیشتر اجزای پردازشگر ساده تر خواهد شد.اما در هر دو صورت مقداری از فضا هدر می رود یا همه روشهای آدرس دهی را نمی تواند پشتیبانی کند!

بار گذاری اعداد صحیح کوچک

- همه ثباتها در MIPS 32 بیتی هستند.
- چه طور ما نیم کلمه یا یک بایت را در یک ثبات بارگذاری می کنیم؟
- بیتها را در حداقل 8یا16بیت ثباتی بارگذاری می کنیم.

بار گذاری بدون علامت: همه بیت های بالا را صفر کنیم.

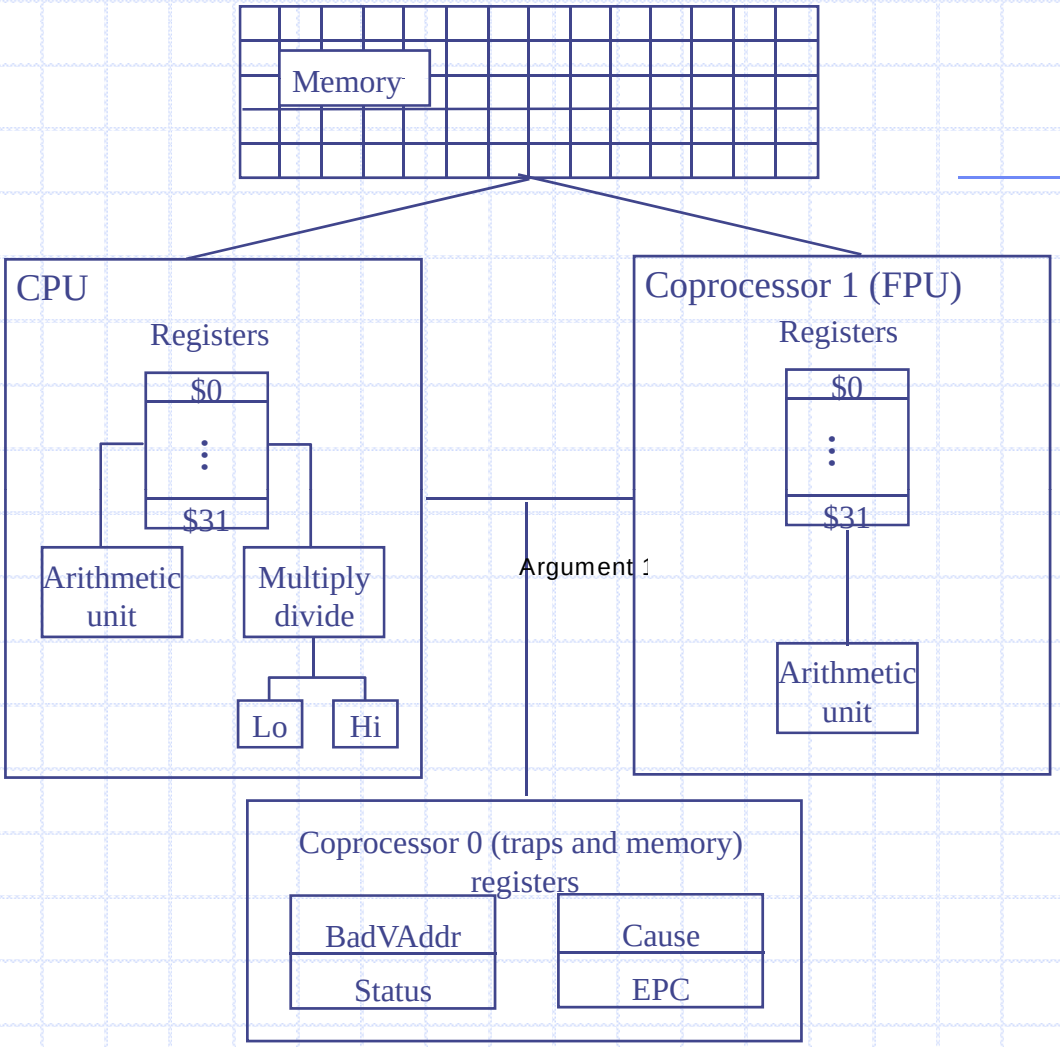
بار گذاری علامت دار: همه بیت های بالا را با توجه به علامت صفر یا یک می کنیم(نیم کلمه یا بایت MSB)

The RISC Approach

- در مجموعه دستورالعمل های کاهش یافته در دستگاه کامپیو تر همه دستورالعمل ها در اندازه های یکسانی هستند (32 بیت در MIPS)
- روشهای آدرس دهی کمی پشتیبانی می شوند (فقط آنهایی که بیشتر کاربرد دارند)
- فقط تعداد کمی از قالب های دستورالعمل(کد برداری آسانتر انجام می شود)
- دستور العمل های محاسباتی که فقط روی ثبات کار می کنند
- داده های حافظه باید در ثبات ها قبل از پردازش شدن بارگذاری شوند که این یک معماری Load_Store نامیده می شود

معیارهای RISC[COL WELL 85]

- عمل تک چرخه ای
- ماشین Load_Store
- کنترل سخت افزاری
- رابطه دستور العمل های کم و روشهای آدرس دهی
- دستورالعمل های با قالب ثابت
- تلاش بیشتر در زمان کامپایل



MIPS R2000 CPU and FPU

ثبات ها

32 تا ثبات باR0=0

ثبات های ذخیره شده:R1,R26,R27.

- کاربرد مخصوص:

اشاره گر به ناحیه سراسری:R28

اشاره گر پشته:R29

اشاره گر قاب:R30

آدرس بازگشت:R31

قراردادهای ثبات استاندارد

- 32 تا ثبات صحیح در MIPSعام منظوره هستند_هر کدام می توانند به عنوان یک عملوند یا نتیجه یک محاسبه عملیاتی استفاده شوند
- اما ساخت تکه های متفاوت از نرم افزارکه باهم کار می کنند آسانتر است اگر قراردادهای معینی که دنبال می شود درباره آن ثباتهایی باشد که برای آن اهداف استفاده می شود
- این قراردادها معمولا به وسیله فروشنده پیشنهاد می شوند و به وسیله کامپایلرها پشتیبانی می شوند

قراردادهای ثباتی در MIPS

Names	Regs	Purpose
$zero	0	Constant 0
-	1	(Reserved for assembler)
$v0-$v1	2-3	Return values/expression eval
$a0-$a3	4-7	Args to functions
$t0-$t9	8-15, 24-25	Temporaries (**NOT SAVED**)
$s0-$s7	16-23	Saved values
-	26-27	(Reserved for OS kernel)
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

MIPS registers and usage convention

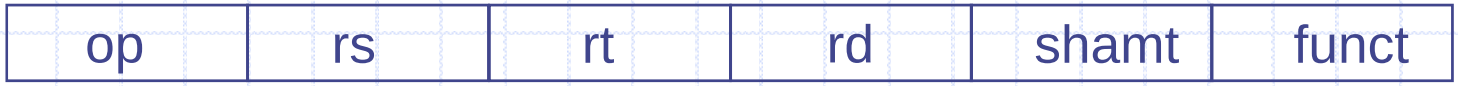
Register name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and results of a function
v1	3	Expression evaluation and results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved aross call)
t1	9	Temporary (not preserved aross call)
t2	10	Temporary (not preserved aross call)
t3	11	Temporary (not preserved aross call)
t4	12	Temporary (not preserved aross call)
t5	13	Temporary (not preserved aross call)
t6	14	Temporary (not preserved aross call)
t7	15	Temporary (not preserved aross call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved aross call)
t9	25	Temporary (not preserved aross call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

عملیات MIPS

- بارگذاری/ذخیره
- عملیات ALU
- انشعاب ها/پرش ها

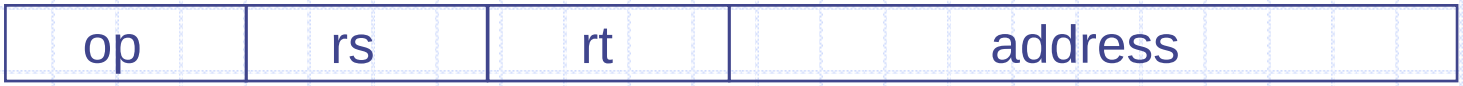
قالب های دستورالعمل MIPS

R-Format



6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

I-Format



6 bits 5 bits 5 bits 16 bits

J-Format

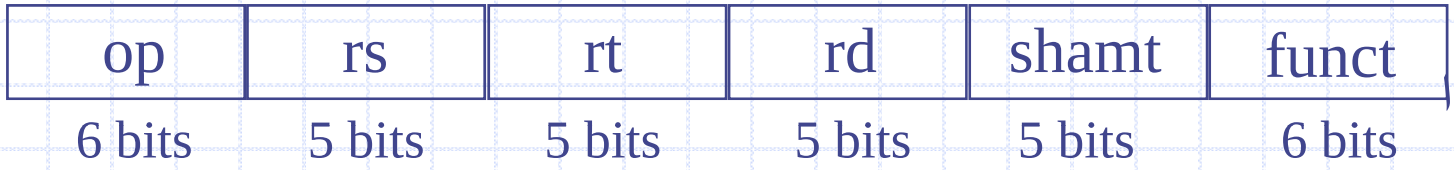
6 bits 26 bits

فیلدها در دستورالعملهای MIPS

- op:مشخصات عمل: بیان اینکه کدام قالب استفاده شود
- rs:اولین ثبات منبع
- rt:دومین ثبات منبع
- rd:ثبات مقصد
- shamt:مقدار شیفت
- funct:جزئیات اضافی opcode
- address:ثابت فوری وتغییرمکان یافته و یا انشعاب

ماشین نمایش دستورالعملهای MIPS

فیلدهای MIPSنام گذاری می شوند که بحث در مورد آنها آسانتر شود:



ینجا منظور هر نامی از فیلدهای دستورالعمل های MIPSاست:

- OP:عملگر هر دستورالعمل
- RS:اولین عملوند ثبات منبع
- RT:دومین عملوند ثبات منبع
- RD:عملوند ثبات مقصد.آن نتایج عملیات را فراهم می کند.
- SHAMT:مقدار شیفت
- FUNCT:تابع:این فیلد عملیات مختلف در فیلد OP را انتخاب می کند.

عملوندهای ALU

R_1, R_2, R_3 ADD

effect: $R_1 = R_2 + R_3$

مثال(شکل اسمبلر در)MIPS

ADD $t1, $s1, $s2

نمایش دهدهی: 0 17 18 8 0 32

نمایش ماشینی

- نمایش دهدهی

0	17	18	8	0	32
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- نمایش باینری

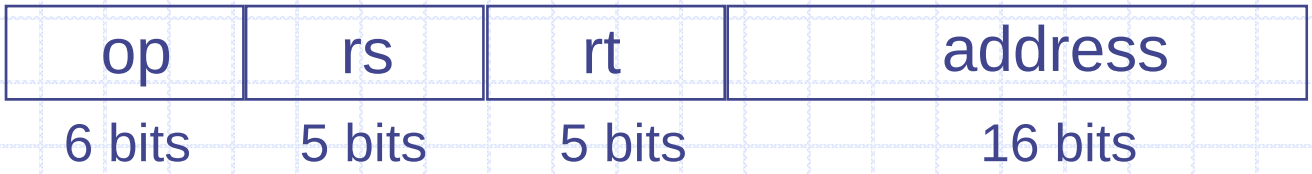
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

ضرب و تقسیم صحیح در MIPS

- ضرب 2 تا عدد 32 بیتی تاحدود 64 بیت می تواند باشد
- 2MIPS تا مخصوص دارد :HIوLO
- نتایج ضرب :بیت های پایین تربه Loمی روند و بیت های بالاتر به Hi می روند
- نتایج تقسیم:خارج قسمت به lo می رود و باقی مانده به hi می رود
- استفاده کردن از عملیات اضافی(از قبیل mflo)برای جابجایی به بالا و پایین ثبات های منظوره

دستورالعملهای انتقال داده ها

I-type (base + 16 bit offsets)



مثال:

lw t0, 8 ($s3) --- # Temporary reg t0 gets A[*8*]

یادداشت:

همچنینrsثبات پایه است($s3در این نمونه _همچنین ثبات شاخص نامیده می شود) و rt (در این نمونه $t0 نتایج را ذخیره می کند (به عنوان ثبات مقصد)

An Example

A=(B+C)*(D+E)انجام می دهدMIPS

Assembly	op	rs	rt	rd	sh.	Ft.
1w $8, 48($0)	35	0	8		48	
1w $9, 76($0)	35	0	9		76	
add $8, $8, $9	0	8	9	8	0	32
1w $9, 20($0)	35	0	9		20	
1w $10, 32($0)	35	0	10		32	
add $9, $9, $10	0	9	10	9	0	32
add $8, $8, $9	0	8	9	8	0	32
sw $8, 100($0)	43	0	8		100	

انشعاب ها

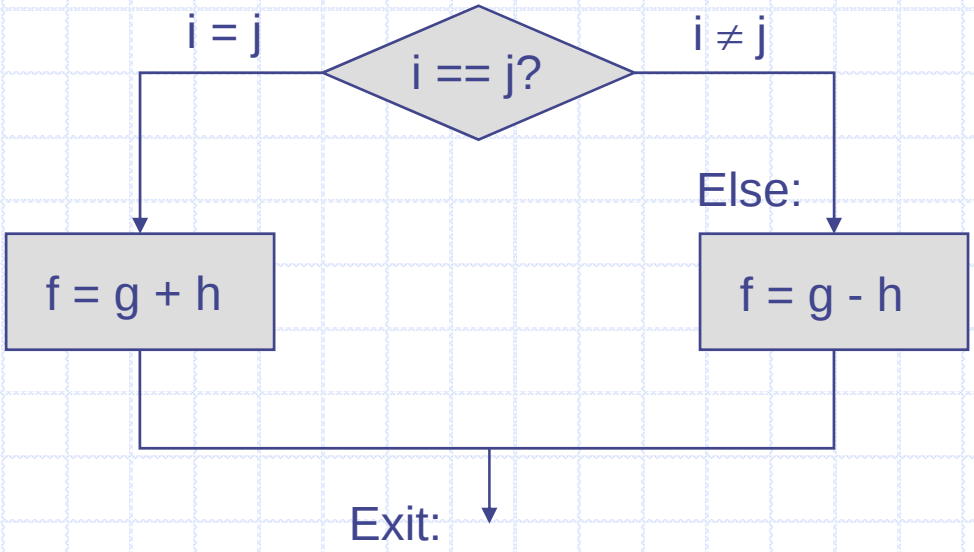
- در بیشترپردازشگرها"شمارنده برنامه"(PC)آدرس دستورالعمل بعدی را نگه می دارد:واکشی از[M[(PC)
- به طور عادی بعداز اینکه یک دستورالعمل تمام شدCPU n تا به PC اضافه می کند n تعداد بایت ها در دستورالعمل است
- انشعاب هابه یک برنامه اجازه می دهند واکشی کنند از مکانهای متفاوت
- انشعابها استفاده میشوند برای به کاربردن همه روند کنترلی فرمانهای زبان های سطح بالا ازقبیل if-then-else, for, switch, etc .

طبقه بندی انشعاب ها

- دو نوع اساسی از پرشها:
- غیر شرطی همیشه به آدرس مشخص شده جهش می کند
- شرطی:اگر شرط درست باشد به آدرس مشخص شده می رود. به عبارت دیگربا دستورالعمل بعدی ادامه می دهد.
- آدرس های مقصد با روش مشابهی می تواند مشخص شود به عنوان عملوندهای دیگر (جمع آوری ثباتها ،ثابت های فوری ، و مکان های حافظه)بستگی دارد که چه چیزی در ISAپشتیبانی شود

مثال همگردانی انشعاب

- کامپایل زیر را دنبال کنید




```
if ( i == j)
  f = g + h;
else
  f = g – h;
```

MIPS در If Then Else

Assume f,g,h,i,j in R8-R12 (respectively)

```
# Branch if i<>j          bne  $11, $12, Else
      # f = g + h;             add $8, $9, $10
   # Jump to Exit                    j Exit
# f = g – h;                sub $8, $9, $10  Else:
   # Code after if                  …   Exit:
```

بررسی انشعاب ها

- بیشتر انشعابهای شرطی برای آدرس های ثابت وکوتاه روی می دهند
- اما اغلب این طور استفاده نمی شوند
- استفاده ای برای افزایش یا کاهش پیشوندی ندارند

پس مطابق با فلسفه سادگی MIPS,RISCتعداد کمی انشعاب اساسی دارد

انواع انشعاب درMIPS

- انشعاب شرطی:beq/bne reg1, reg2, addr

  - If reg1 =/≠ reg2, jump to PC + addr (PC-relative)

- پرش ثباتی:jr reg

واکشی آدرس از ثبات مشخص شده و پرش به آن

- پرش غیر شرطی:j addr
- همیشه به آدرس پرش می کند(استفاده “pseudododirect”آدرس دهی)

دستورالعمل های انشعاب

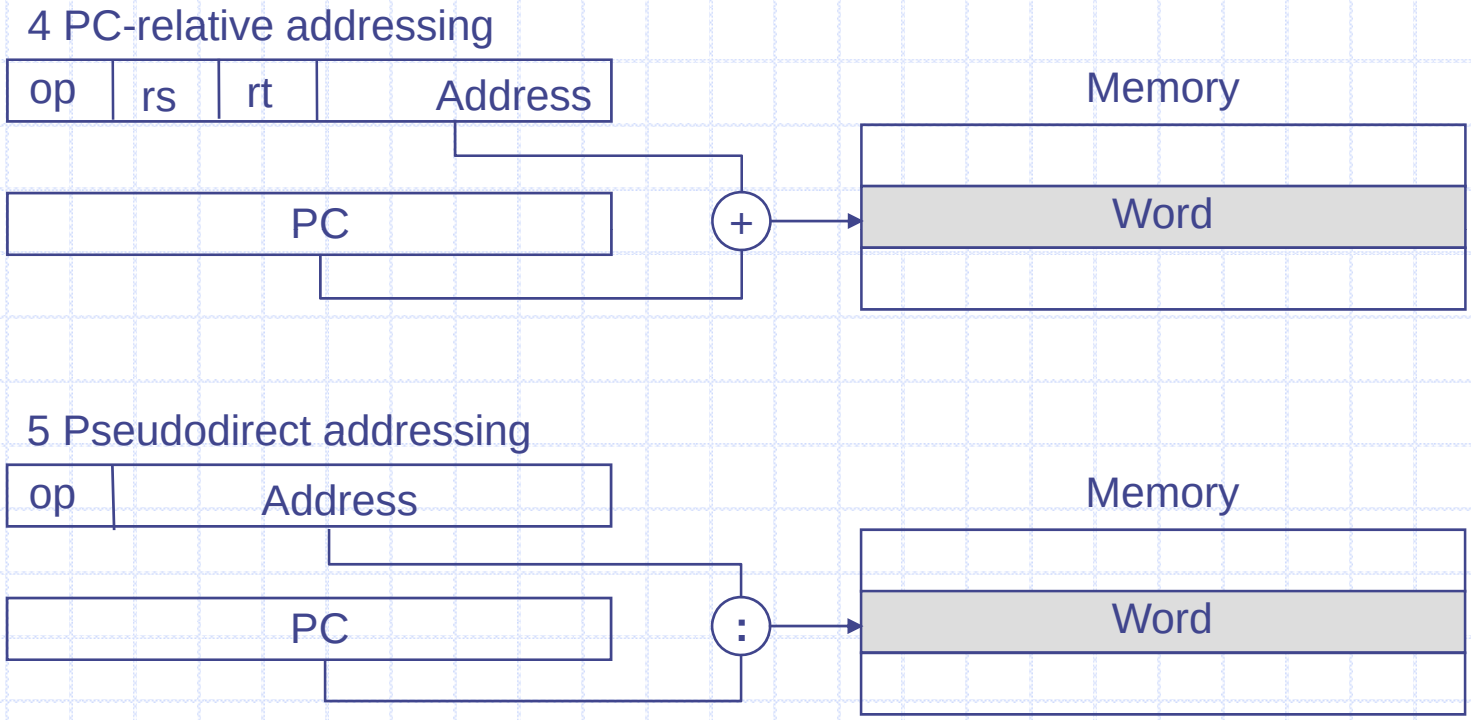
- انشعاب های شرطی

- beq R1, R2, L1 # if R1 = R2 go to L1
- bne R1, R2, L1 # if R1 =\= R2 go to L1

- این ها دستورالعملهای R_type هستند
- انشعاب های غیر شرطی

```
JR     R8    #  Jump based on register 8
Test if < 0
            slt    R1, R16, R17     # R1 gets 1 if  R16 <  R17
                                    (slt: set-less-than)
            bne   R1, 0, less       # branch to less if R1 =\= 0
```

مولد اهداف انشعاب در MIPS



مثال تلفیق سوئیچ

- کامپایل را دنبال کنید:

```
switch    (k)    {
              case 0:  f  =  f +
1;       break;
              case 1:  f  =  f –
2;       break;
              case 3:  f  = -f;
         break;
         }
Note the gap (case 2);
```

بدنه سوئیچ در MIPS

```
1  to  r8   (f) immed.  $8, $8, 1    add      addi   L0:
    jump to Exit  (break)                  j Exit
$8, $8, 2    subtract imm.   2 from   r8      subi      L1:
    Another break                  j Exit
    f  =  0 -  f sub $8,  $0,  $8        L3:
    Another break                  j Exit
```

ساختن جدول مراجعه درحافظه

1000	address of L0
1004	address of L1
1008	address of Exit
1012	address of L3

سوئیچ کامپایل شده برای MIPS

(Assume k in r13)

```
slti  $14,  $13,  0      # set r14  if r13  1t 0
bne   $14, $0, Exit      # Go to Exit if k < 0
slti  $14,  $13,  4      # set r14  if k < 4
beq $14,  $0, Exit       # Go to Exit if k ≥ 4
add $14,  $13, $13       # r14 = 2*k
add $14,  $14, $14       # r14 = 4*k
lw $14, 1000 ($14)       # Base of table at 1000
jr   $14                 # Jump to the address
```

کامپایل کردن دستورات کنترلی دیگر

- حلقه ها:
- For,while:تست کردن قبل از بدنه حلقه. پرش کن به بعد از بدنه حلقه اگر شرط نادرست است .
- Do:تست کردن شرط در انتهای بدنه حلقه.پرش کن به ابتدا اگر درست است
- سوئیچ:(“case” فراخوانده می شوددر بعضی زبانهای دیگر)
- ساختن یک جدول آدرس
- استفادهjr(or equiv. In non-MIPS processor)
- مطمئن باشید از چک کردن برای پیش فرض یا caseهای بدون استفاده

پشتیبانی دستورالعملهای فراخونی پردازه

Jump and link

آدرس پردازشگر jal

یادداشت:

آدرس بازگشت در r31ذخیره می شود

Return

jr R31

ذخیره کردن آدرس بازگشت روی پشته

R29 به عنوان اشاره گر پشته استفاده می شود

پارامتر زود گذر:

R_4 ~ R_7استفاده می شوند برای اینها

روش آدرس دهی MIPSهای دیگر

- عملوندهای ثابت یا فوری

LW R24, AddrConstant4(0)

Addi R3, 5 (I type)

ثابتها16بیتی هستند

Lui R8 255 Load_upper_immediate

- می رود به مکان 10000

j_type J 10000

عملوندهای MIPS

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp (28) is the global pointer, $sp(29) is the stack pointer, $fp (30) is the frame pointer, and $ra (31) is the return address.
2^{30} memory words	Memory [0], Memory [4],…, Memory[42949672920	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled register, such as those saved on procedure calls.

زبان اسمبلی MIPS

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
Data transfer	load word	lw $s1,100 ($s2)	&s1 = Memory [$s2 + 100]	Data from memory to register
	store word	sw $s1,100 ($s2)	Memory [$s2 + 100] = $s1	Data from register to memory
Conditional branch	branch on equal	beq $s1, $s2, L	if ($s1 == $s2) go to L	Equal test and branch
	branch on not equal	bne $s1, $s2, L	if ($s1 != $s2) go to L	Not equal test and branch
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than: for beq, bne
Unconditional jump	jump	j 2500	go to 10000	jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 1000	For procedure call

زبان ماشین MIPS

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
lw	I	35	18	17	100			lw $s1,100 ($s2)
sw	I	43	18	17	100			sw $s1,100 ($s2)
beq	I	4	17	18	25			beq $s1, $s2, 100
bne	I	5	17	18	25			bne $s1, $s2, 100
slt	R	0	18	19	17	0	42	slt $s1, $s2, $s3
j	J	2	2500					j 10000 (see section 3.8)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000 (see section 3.8)
field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructins 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

فراخوانی تابع درMIPS

- فراخوانی تابع یک ساختار ضروری زبان های برنامه نویسی است.

برنامه یک تابع را برای انجام چند وظیفه فرامی خواند.

موقعی که تابع انجام میشودCPU دقیقا از بعد از جایی که برنامه فراخوانی شده ادامه می دهد

فراخوانی تابع در MIPS

استفاده از دستورالعمل Jal(“jump and link”)

Jal addr فقط J adrr است به جز موارد زیر

“آدرس بازگشت” (pc)+4placed in R31

این آدرس دستورالعمل بعدی بعد ازjalاست

از jr$31برای بازگشت استفاده می کنیم

مثال فراخوانی

Callee

```
F:  lw  $6,  0($4)
    lw  $7,  0($5)
    sw  $6,  0($5)
    sw  $7,  0($4)
    jr    $31
```

Caller

```
        add  $4,  $0, 1000
        add  $5,  $0, 1200
        add  $1,  $0, 1
    sw   $1,  0($4)
        add $1,  $1, $
sw   $1, 0 ($5)
        jal   F
sub  $1, $1, $2
```

What does F do?

مشکلات فراخوانی تابع ها

این مثال درست کار می کند اما چه طور؟

تابع F تابع دیگر را فراخوانی می کند

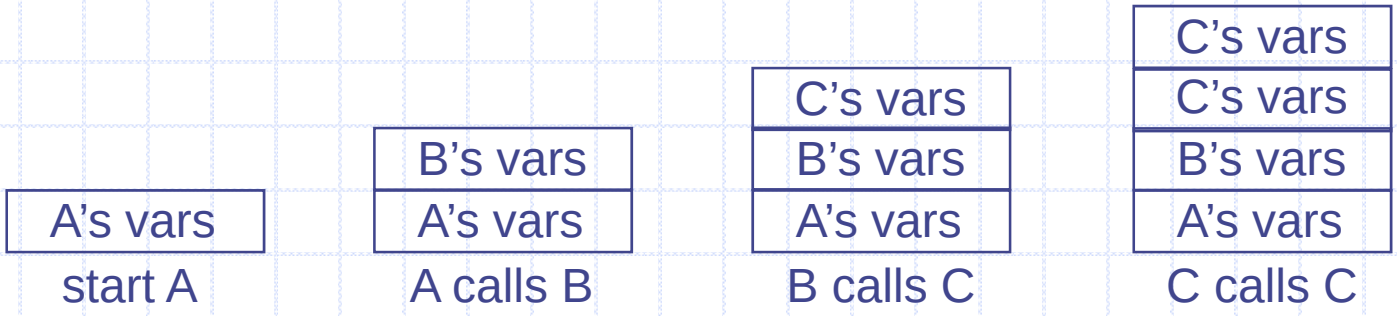
فراخوانی کننده یک نکته مهم در ثباتهای R6وR7دارد؟

تابع فراخوانی شده خودش را فرامی خواند؟

هر نسخه از یک تابع باید کپی متغیر هایش را داشته باشداین ها در یک پشته به عنوان یک ستون از قاب ها مرتب شده اند.

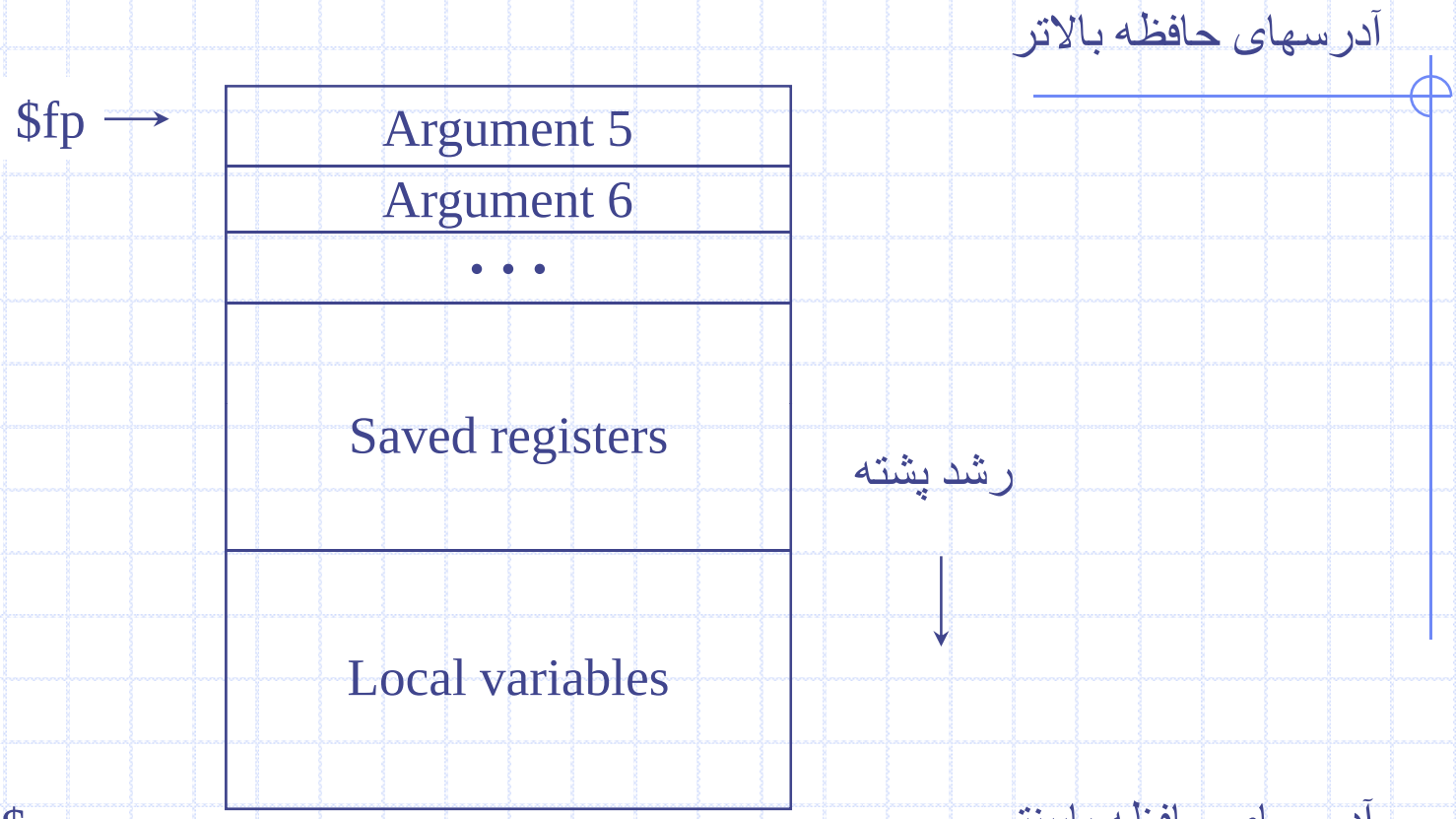
مثال پشته

فرض کنید تابع A, Bرا فراخوانی کند و C, B را فراخوانی کند.تابع Cخودش را فراخوانی می کند.



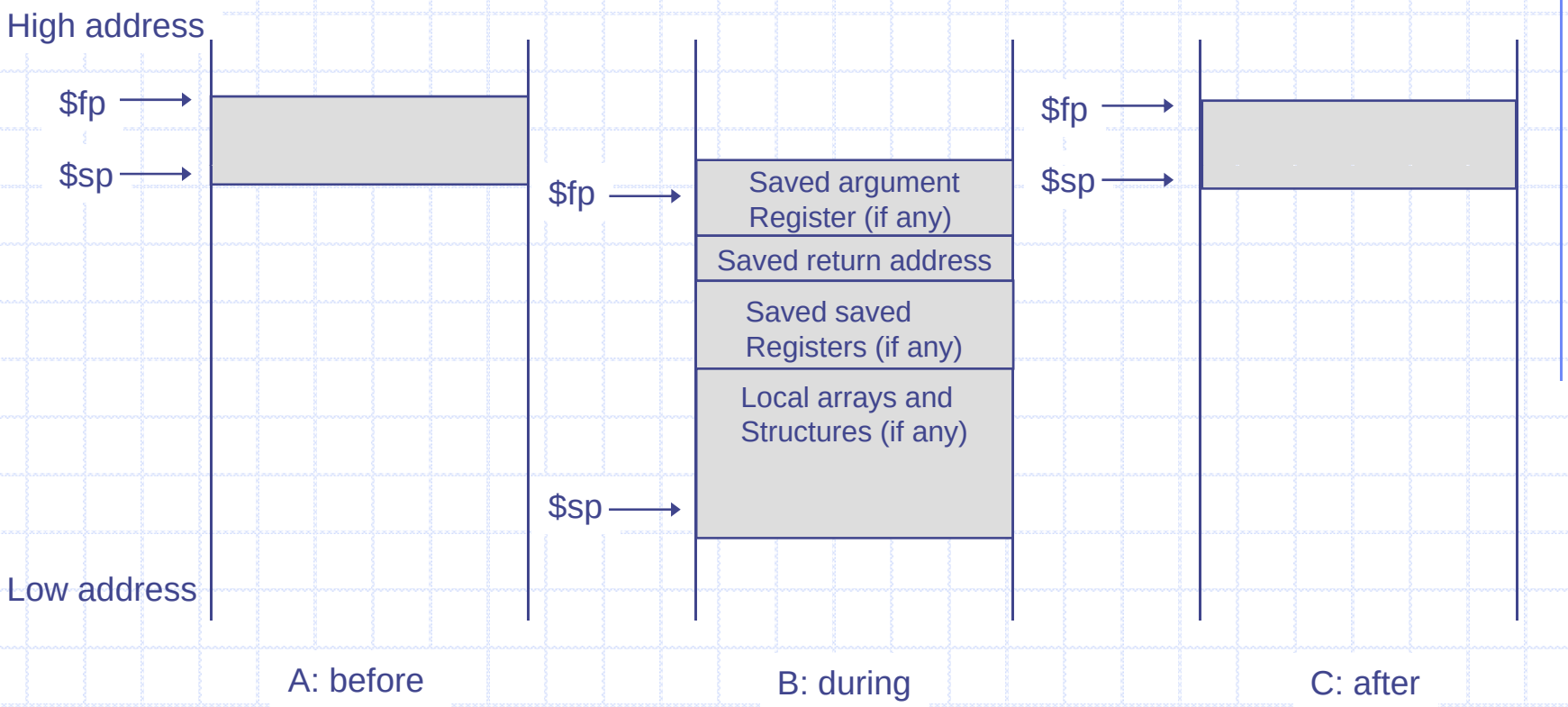
فراخوانی پردازه

- مکان پارامترها
- کنترل انتقال
- ذخیره سازی Acquire
- انجام دادن کار
- مکان بازگشت نتایج
- بازگشت کنترل بر گشتی به فراخوان



آرایش قاب پشته

جزئیات پشته



قراردادهای ثبات استاندارد

- 32 تا ثبات صحیح در MIPSعام منظوره هستند_هر کدام می توانند به عنوان یک عملوند یا نتیجه یک محاسبه عملیاتی استفاده شوند
- اما ساخت تکه های متفاوت از نرم افزارکه باهم کار می کنند آسانتر است اگر قراردادهای معینی که دنبال می شود درباره آن ثباتهایی باشد که برای آن اهداف استفاده می شود
- این قراردادها معمولا به وسیله فروشنده پیشنهاد می شوند و به وسیله کامپایلرها پشتیبانی می شوند

پارامترهای میانی

- ذخیره کردن فراخوانی کننده:پردازه فراخوان(caller)مسئول است برای ذخیره کردن و دوباره ذخیره کردن هر کدام از ثباتهایی که باید در طی فراخوانی نگهداری شوند.پردازه فراخوانی شده (callee)می تواند هر ثباتی را بدون محدودیت تغییر دهد.
- ذخیره کردن فراخوانی شده: پردازه ای که فراخوانی شده مسئول است برای ذخیره کردن و دوباره ذخیره کردن هر کدام از ثباتهایی که ممکن است استفاده شوند.

فرخوان ثباتها را بدون هیچ نگرانی درباره دوباره ذخیره کردن آنها بعد از یک فراخوانی استفاده می کند.

ثباتهای ذخیره سازی

- اگر شما یک تابع را فراخوانی کنید هر آنچه شما در $S0تا$S7دارید در آنجا تضمین شده زمانی که تابع برمی گردد به شما
- اما ثباتهای $t0 تا $t9 fair gameهستند به وسیله تابع دوباره استفاده می شود
- چه گزینش هایی هستند؟
- هیچ ذخیره کردن؟
- همه چیز ذخیره کردن؟

اعلام یک تابع در MIPS

قرار بدهید آرگومانهای تابع را در $a0_$a3
اگر یک چیز مهم در $t0_9باشد اکنون آنها را انتقال می دهد!
jal به تابع
ذخیره کردن $s0_$s7و$raاگرلازم باشد شما کاری انجام دهید
و مقادیر بازگشت را(اگر داشته باشد) در $v0_$v1
دوباره ذخیره میکنیم $s0_$s7و$raرااگرلزومی داشته باشد
ادامه دهید تا جایی که تمام شود
سوال: چه کسی آن را به عنوان یک طرح می سازد؟
چه اتفاقی می افتد اگر ثباتهای ما تمام شود؟
واقعا چه چیزی عمل ذخیره کردن را انجام می دهد؟

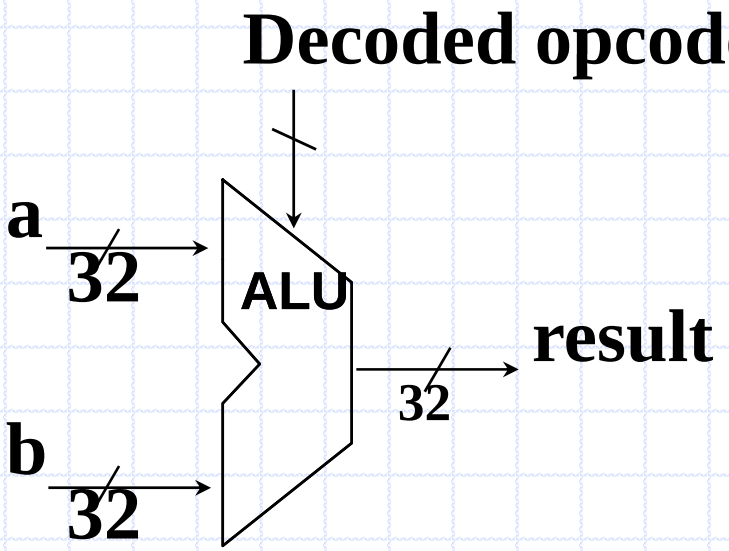
فصل سوم
نمایش اعداد و محاسباتی کامپیوتر

چکیده

- مقدمه
- نمایش عددی و مکمل دو
- طراحی واحد محاسبه ,منطق
- ضرب و تقسیم صحیح
- اعداد اعشاری

مرور

- بازبینی
  - مراحل اجرای یک دستورالعمل
- فیلدهای یک دستورالعمل
  - کارایی؟
  - مبانی عملوندها؟
- آنچه پیش روی ماست:
  - نمایش عددی
  - الگوریتم های محاسباتی
  - پیاده سازی سخت افزاری
  - دستورالعملها



نمایش عددی

- آنچه پیچیده به نظر می رسد:
  - اعداد منفی را چگونه نمایش دهیم
  - دامنه (Range) اعداد چیست؟
  - اگر یک عدد خارج از محدوده بود؟
  - در مورد اعداد منطقی و غیر منطقی چطور؟
  - سخت افزار چگونه این اعداد را جمع، تفریق،ضرب و تقسیم میکند؟

بیت ها و اعداد

- بیتها فقط بیت هستند: هیچ معنای ذاتی ندارند.
  - قراردادها رابطه بین بیتها و اعداد را تعیین می کنند.

  - سخت افزار چگونه تشخیص می دهد که چه قراردادی در حال استفاده است؟

Add $s1, $s2, $s3

Addu $s1, $s2, $s3

علامت $s1،$s2 و $s3 چیست؟

نمایش عددی

- صحیح بدون علامت
- صحیح با علامت
- BCD دهدهی کد شده به باینری
- ممیز ثابت
- ممیز شناور
- انواع دیگر داده:
  - کاراکترها(اسکی و یونیکد)
  - پیکسل ها (گرافیک ها)
  - گروهی از بیتها

صحیح بدون علامت

- چرا صحیح بدون علامت؟
  - دستیابی به حافظه PC, SP, RA,
  - In C, unsigned int
- چگونه نمایش می دهیم؟
  - انگشتهای خود را بشمارید.
  - مبنای اولیه سیستم اعداد
    - دهدهی: 10 نماد مختلف: 0 1 2 3 4 5 6 7 8 9

$$\text{عدد} = \sum_{i=0}^{n-1} \left(d_i * 10^i \right)$$

صحیح بدون علامت (مبنای اولیه سیستم اعداد)

- در حالت کلی، در یک سیستم عددی مبنای K

$$\text{عدد} = \sum_{i=0}^{n-1} (d_i * k^i)$$

  - چه تعداد d_i متفاوت؟
  - بزرگترین عدد کدامست؟
  - کوچکترین عدد کدامست؟

- مبناهای خاص:
  - مبنای 2 (Binary)
  - مبنای 8 (octal)
  - مبنای 16(hexademical)
    - آنها چه تعداد نماد مختلف دارند؟

صحیح بدون علامت (مبنای اولیه سیستم اعداد)

- چگونه می توان مبنای دو را به مبنای 8 و 16 تبدیل کرد؟
- مثال:

$$010100=(0*2^2+1*2^1+0*2^0)*2^3+(1*2^2+0*2^1+0*2^0)$$
$$=2*8^1+4*8^0=24_8$$

Hexadecimal:	1	9	4	8	B	6	
Binary:	000	110	010	010	001	011	0
Octal:	0	6	2	4	4	2	6	6

صحیح بدون علامت (BCD: binary coded decimal)

- نمایش : 127_{10}
  - چه تعداد بیت برای BCD مورد نیاز است؟ برای دودویی چطور؟
  - فضای ذخیره سازی بکارگرفته شده؟
- ضمیمه:
  - چگونه می توان carry را مشخص نمود؟

  "If $a_i + b_i > 10$" vs. "If $a_i + b_i > 2$"
- کدامیک برای کامپیوتر کاراتر است؟

BCD	Number
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010-1111	No use

صحیح با علامت (علامت مقدار)

- 12(absolute value) -(sign) :-12
  - یک بیت علامت مجزا
  - یک مقدار
- برای سخت افزار
  - بیت علامت را کجا قرار دهیم؟چپ یا راست؟
  - علامت حاصلجمع را چگونه تعیین می کنیم؟ (مثلا یک مرحله اضافی)
  - علامت صفر چیست؟ مثبت یا منفی؟

مکمل 2

اعداد و نمایش آنها یکی نیست.

- نمایش عدد، یک **قرارداد** تعریف شده برای بیان آن است.
- یک عدد می تواند دارای نمایشهای متفاوتی باشد مثلا BCD مبنای 16، مکمل دو و غیره.
- مکمل دو: یک نوع از نمایش که بصورت زیر تعریف شده است:

نمایش	عدد
$b_{n-1}\,b_{n-2}\ldots\ldots b_1 b_0$	$b_{n-1} * (-2^{n-1}) + \sum_{i=0}^{n-2} (b_i * 2^i)$

مكمل دو

$$b_{n-1} * (-2^{n-1}) + \sum_{i=0}^{n-2} (b_i * 2^i)$$

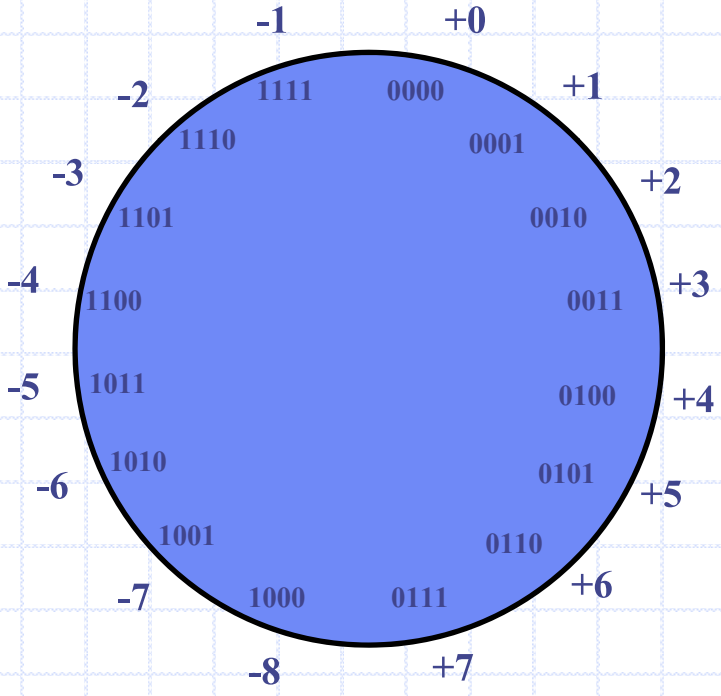
$b_{n-1}\, b_{n-2} \ldots\ldots b_1 b_0$

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

صحیح با علامت(مکمل دو)

(برای 4 بیت)



مرجع: Katz: Contemporary Logic Design, p243

چرا 1111 و 0000 همسایه اند؟

مفهوم بیت سوم چیست؟

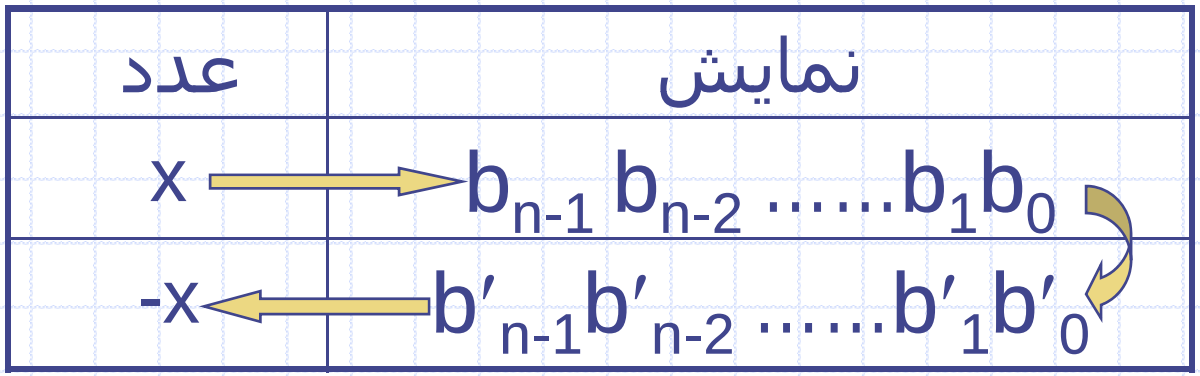
علامت صفر چیست؟

کوچکترین عدد کدامست؟

بزرگترین عدد کدامست؟

مکمل دو

- نمایش یک عدد منفی x-
  - از نمایش عدد X تا نمایش x-



هر bi را معکوس کرده، سپس با 1 جمع کنید.

- اثبات درستی

- **میانبر:** از با ارزشترین بیت تا کم ارزشترین 1هر بیت را معکوس کنید و کم ارزشترین یک را 1 نگاه دارید.

Example: - (0100) = 1011 + 1 = 1100

Example: 100_{10} = 01100100 → 10011100 = -100_{10}

گنجاندن اعداد کوچکتر در بیتهای بیشتر

- چرا لازم است؟
  - مقایسه یک integer با یک long integer : تغییر نوع
  - بارگذاری یک بایت در یک کلمه
  - برای اضافه کردن یک بخش فوری به یک عدد 32 بیتی

- واحد محاسبه و منطق MIPS فقط با مقادیر موجود در رجیسترهای 32 بیتی کار می کند.
  - چگونه می توان با اندازه های کوچکتر کار کرد؟
  - در مورد اندازه های بزرگتر چطور؟

گنجاندن اعداد کوچکتر در بیتهای بیشتر

- Example(2-bit): $10 = 1 *(-2^1)+0 *2^0=-2_{10}$

نمایش 2- مبنای 10 در 4 بیت کلمه: آیا می توانید آنرا انجام دهید؟ چه چیزی کشف می شود؟

- تکرار بیت علامت در بیتهای دیگر(گسترش علامت)

0010 -> 0000 0010

1010 -> 1111 1010

(? = 111 1010 + (2^(-7))x1)نکته! اثبات ؟

جمع و تفریق

- جمع : درست مانند دوران شیرین دبستان

0110
+ 0001

- تفریق : a-b=a+(-b)
  - اعمال مکمل دو ساده است

0111-0110= 0111
+1010
1 1
10001

  - بنابراین 0111-0110= 0001
  - آیا شما مخصوصا بیتهای علامت را دستکاری نموده اید؟

جمع و تفریق

- صبر کنید! یک بیت اضافی گم شده است!

0111-0110= 0111

چرا 1 می تواند حذف شود؟ 1010+

1 1

10001

- زیرا

is

$0*2^3$

$+ 1*(-2^3)$

$1*2^3$

0

اما رفتار می کند در جمع مانند

$0*2^3$

$+ 1*2^3$

$1*2^3$

10

- از دیدگاه 1 بیت : 1-1=1+1=0
- بنابراین تا زمانیکه شما carry بوجود آمده بوسیله 1+1 (1 سبز) را در نظر نگرفته اید، درست انجام داده اید.

جمع و تفریق

- سئوالات:
  - آیا شما یک جمع کننده و یک تفریق کننده دارید؟
  - در مورد اعداد بدون علامت چطور؟(یک جمع کننده دیگر؟)
- مزایای استفاده از مکمل 2
  - تفریق می تواند از منطق مشترکی با جمع استفاده کند
  - بیت علامت می تواند مانند یک بیت معمولی عدد رفتار کند
  - اینها نکات زیرکانه ای هستند .

سرریزی

- جمع:

```
  1111      0111
+ 0110    +1000
------    -----
```

علامت ورودی چیست؟

علامت خروجی چیست؟

- آیا امکان وقوع سرریزی در جمع یک عدد مثبت و یک عدد منفی وجود دارد؟
- امکان وقوع سرریز با صفر وجود دارد؟
  - اعمال A+B و A-B را ملاحظه کنید.

سرریزی

- جمع

```
  1111      0111
+ 1000    + 0110
------    ------
 10111      1101
```

علامت ورودی چیست؟

علامت خروجی چیست؟

کشف سرریزی

- سرریزی زمانی اتفاق می افتد که:
  - جمع دو عدد منفی یک عدد مثبت بدهد.
  - جمع دو عدد مثبت یک عدد منفی بدهد.
  - یا یک عدد منفی را از مثبت تفریق کنیم و نتیجه منفی بگیریم
  - یا یک عدد مثبت را از منفی تفریق کنیم و نتیجه مثبت بگیریم
- در مورد جمع و تفریق اعداد بدون علامت چطور؟

کار CPU در برابر سرریزی چیست؟

- نادیده بگیرد؟
  - هیچ وقت نمی خواهد سرریزی آشکار شود
  - Addu, addiu, subu (MIPS:عدم تولید سریز)
  - تولید یک تله که بر اساس آن برنامه نویس بتواند با آن کار کند.
  - یک استثنا (وقفه)رخ دهد
    - کنترل به آدرس از پیش تعریف شده استثنا پرش کند
    - ذخیره آدرس مکان وقفه برای امکان بازگشت
    - جمع ، تفریق

دستورالعمل ها

- سنجش
  - اعداد بدون علامت

sltu: set on less than unsigned

sltiu: set on less than immediate unsigned

  - اعداد با علامت

slt: set on less than

slti: set on less than immediate

- مثال: مقادیر $s0 و $s1 چیست؟

$s0= 1111 1111 1111 1111 1111 1111 1111 1111

$s1= 0000 0000 0000 0000 0000 0000 0000 0001

(1) slt $t0, $s0, $s1 (2) sltu $t1, $s0, $s1

مثال

- مقادیر $s0 و $s1 چیست؟

$s0= 1111 1111 1111 1111 1111 1111 1111 1111

$s1= 0000 0000 0000 0000 0000 0000 0000 0001

پاسخ:

-- slt $t0, $s0, $s1 اگر هر دو با علامت باشند $t0 است 1

-- sltu $t1, $s0, $s1 اگر هر دو بدون علامت باشند $t1است 0

دستورالعمل ها

- بارگذاری/ذخیره سازی
  - lb: load byte
  - lbu: load byte unsigned
- Example: lb $s1, 100($s2)

مقادیر $s0 چیست

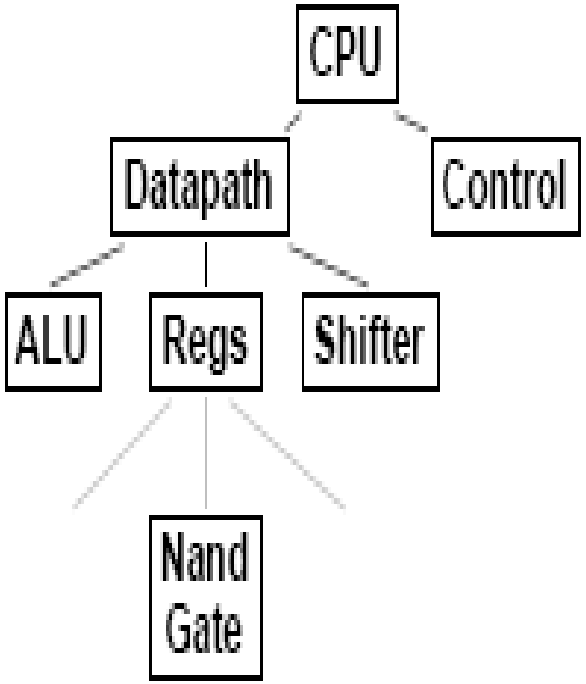
- When memory[$s2+100] = 0000 0000?
- When memory[$s2+100] = 0000 0001?
- When memory[$s2+100] = 1111 1111?

- Example: lbu $s1, 100($s2)

مقادیر $s1 چیست

- when memory[$s2+100] = 1111 1111?

طراحی پردازش



- طراحی
  - اجزا
  - آنها چگونه در کنار هم قرار می گیرند
- تجزیه بالا به پایین بخشهای مختلط
- ترکیب پایین به بالای بخشهای ساده

مسئله: طراحی واحد محاسبه و منطق

- عملیات
  - add, addu, sub, subu, addi, addiu
    - جمع کننده تفریق کننده مکمل 2 با آشکار سازی سرریز
  - and, or, andi, ori
    - اعمال بیتی

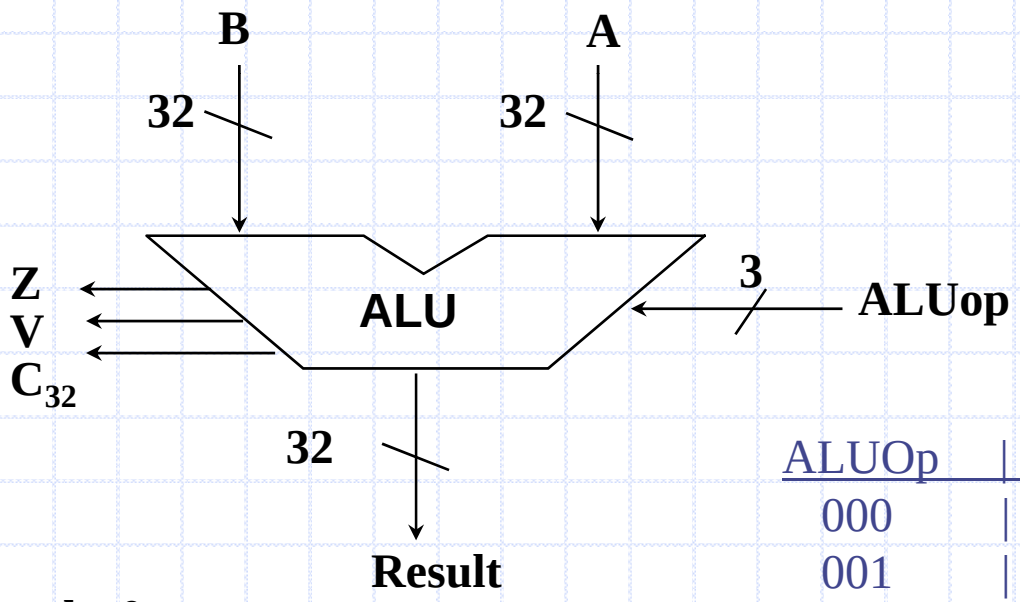
مجموع اعمال = 10

طراحی: روش تقسیم و غلبه

- شکستن مسئله به بخشهای ساده تر
  - کار با اجزا
  - قرار دادن قطعات با یکدیگر
  - بازبینی راهکارها در نهایت
- مثال: جدا کردن دستورات فوری از بقیه دستورات
  - پردازش فوریها قبل از ALU
    - اکنون ورودیهای ALU یک شکل هستند.
  - 6 عمل غیر فوری باقی مانده است
    - نیازمند 3 بیت برای تعیین مد ALU است

محیط ALU (ALU Interface)

- ما یک ALU 32 بیتی طراحی خواهیم کرد با محیط زیر:



Z =1, if Result=0
V= 1, if Overflow
C_{32} = 1, if Carry-Out

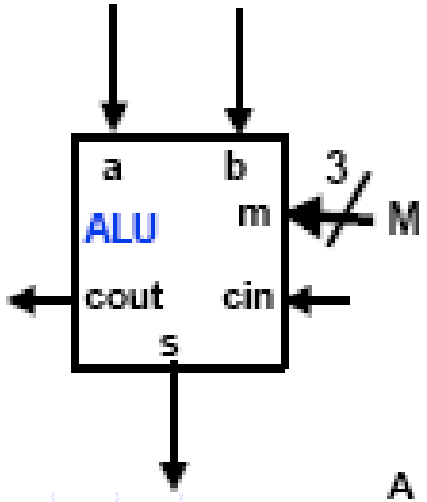
ALUOp	Function
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET-ON-LESS-THAN

طراحی: کاهش مسئله به یک مسئله ساده تر

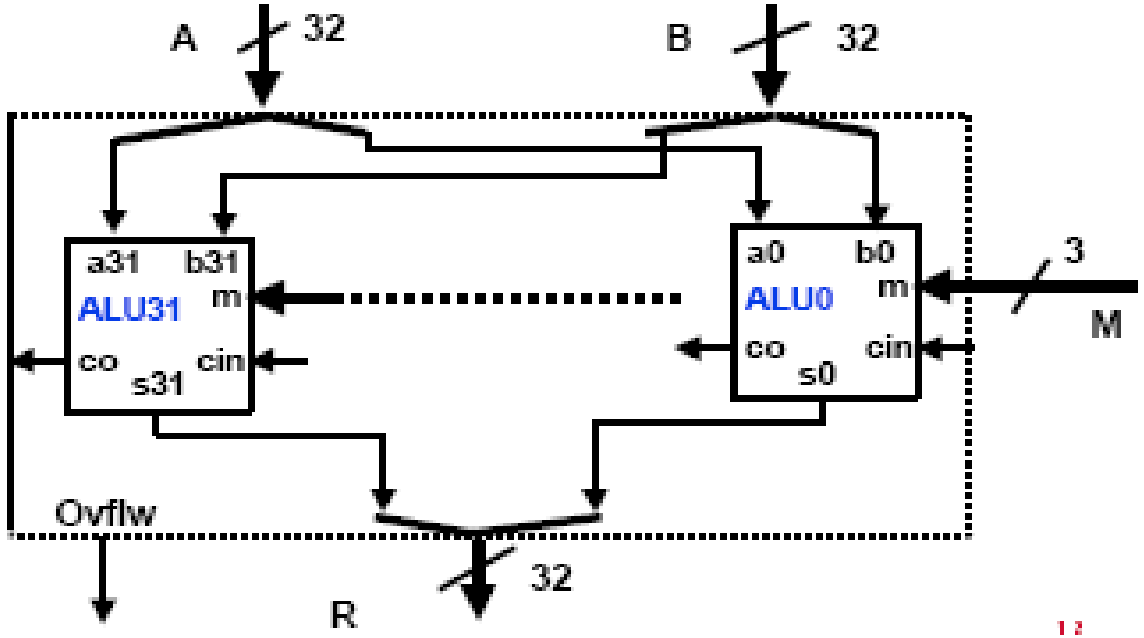
- برای این مسئله، ما ALU 32 بیتی را به بخش ساده تر یک بیتی کاهش می دهیم.
  - مسائل ترکیبی بزرگ را به یک مسئله ترکیبی کوچکتر تغییر می دهیم
- بخشها را برای حل مسئله بزرگ در کنار هم قرار می دهیم.

طراحی با بلاک دیاگرام سطح پایین تر

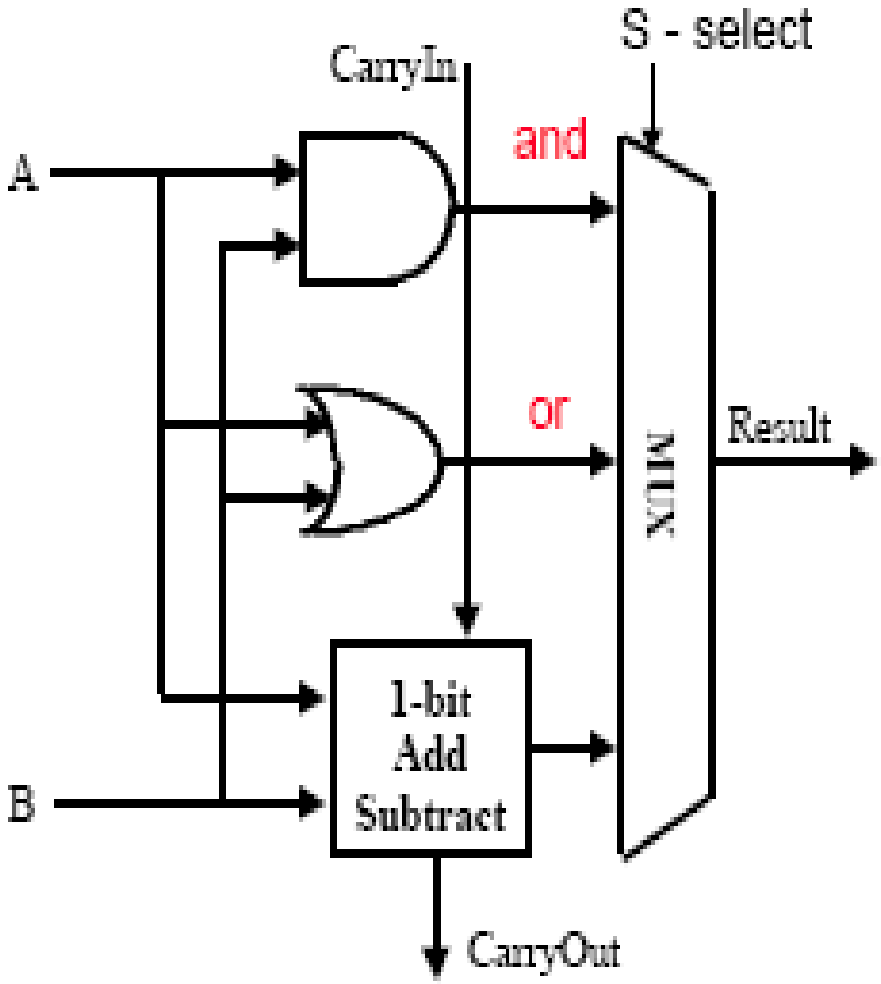
بلاک ALU
یک بیتی



برای یک ALU 32 بیتی
32 بار تکرار می کنیم



بلاک ALU یک بیتی

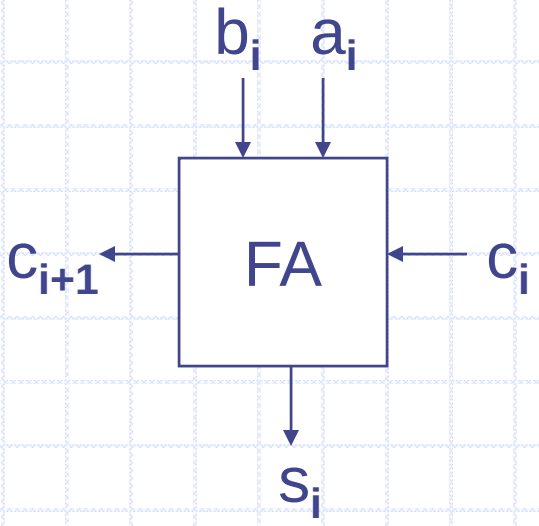


- تقسیم به بلاکهای جدا و مستقل
  - منطق
  - محاسبه
- تکمیل هر بلاک در این مرحله برای تخلیص بیشتر
  - تکمیل بلاکهای منطقی
  - تکمیل انتخاب عمل
  - تجزیه بلاکهای محاسباتی به بخشهای ساده تر

تمام جمع کننده Full Adder

- تمام جمع کننده، یک بلاک ساخته بنیادی در ALU است.
- یک تمام جمع کننده، جمع یک بیت را انجام می دهد.

$$a_i + b_i + c_i = 2c_{i+1} + s_i$$



c_{i+1} c_i	1 1	1 0
a_i	0 1	0 1
b_i	0 1	1 1
s_i	1 1	0 0

جدول درستی تمام جمع کننده

- Si ،"1" است اگر یک، تک عدد از ورودیها یک باشد.
- Ci+1 ،"1" است اگر 2عدد (یا بیشتراز) ورودیها 1 باشند.

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s_i = \bar{a}_i\bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i\bar{b}_i\bar{c}_i + a_i b_i c_i$$
$$s_i = a_i \otimes b_i \otimes c_i$$
$$c_{i+1} = \bar{a}_i b_i c_i + a_i\bar{b}_i c_i + a_i b_i\bar{c}_i + a_i b_i c_i$$
$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$
$$c_{i+1} = a_i b_i + c_i(a_i + b_i)$$
$$c_{i+1} = a_i b_i + c_i(a_i \otimes b_i)$$

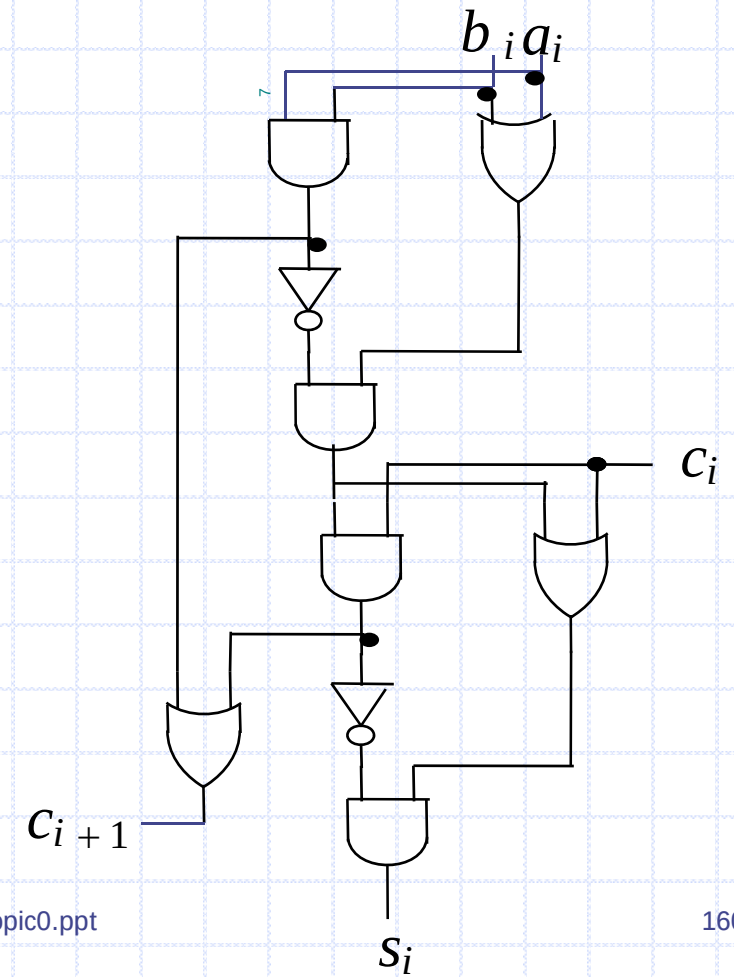
طراحی تمام جمع کننده

- یک پیاده سازی ممکن از یک تمام جمع کننده که gate 9 بکار برده است

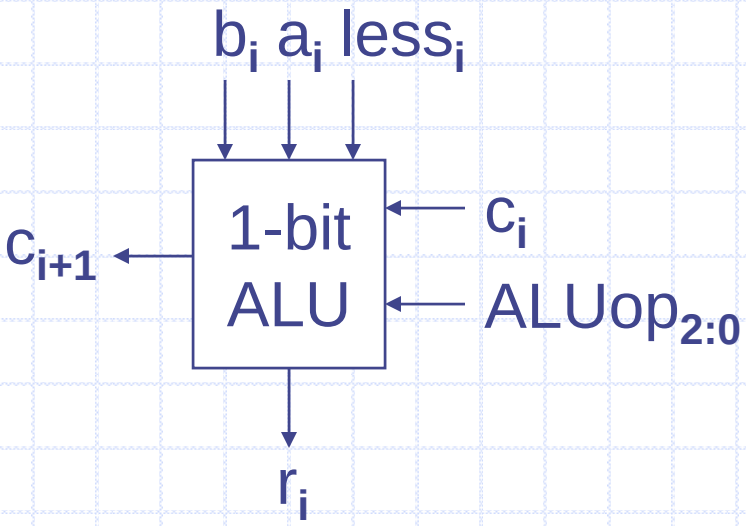
$$s_i = a_i \otimes b_i \otimes c_i$$

$$c_{i+1} = a_i b_i + c_i(a_i \otimes b_i)$$

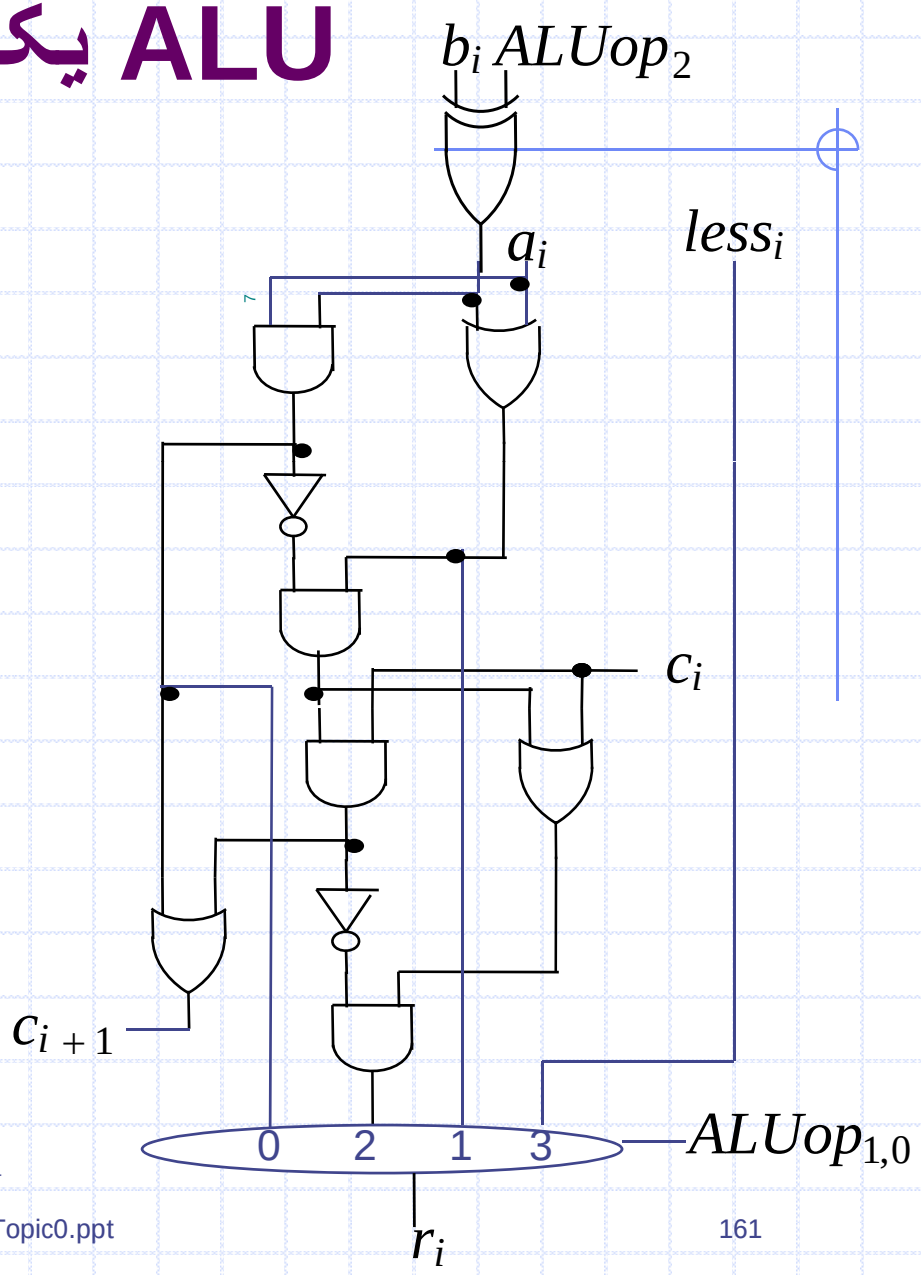
$$a_i \otimes b_i = (a_i + b_i)\overline{a_i b_i}$$



ALU یک بیتی



ALUOp	Function
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET-ON-LESS-THAN

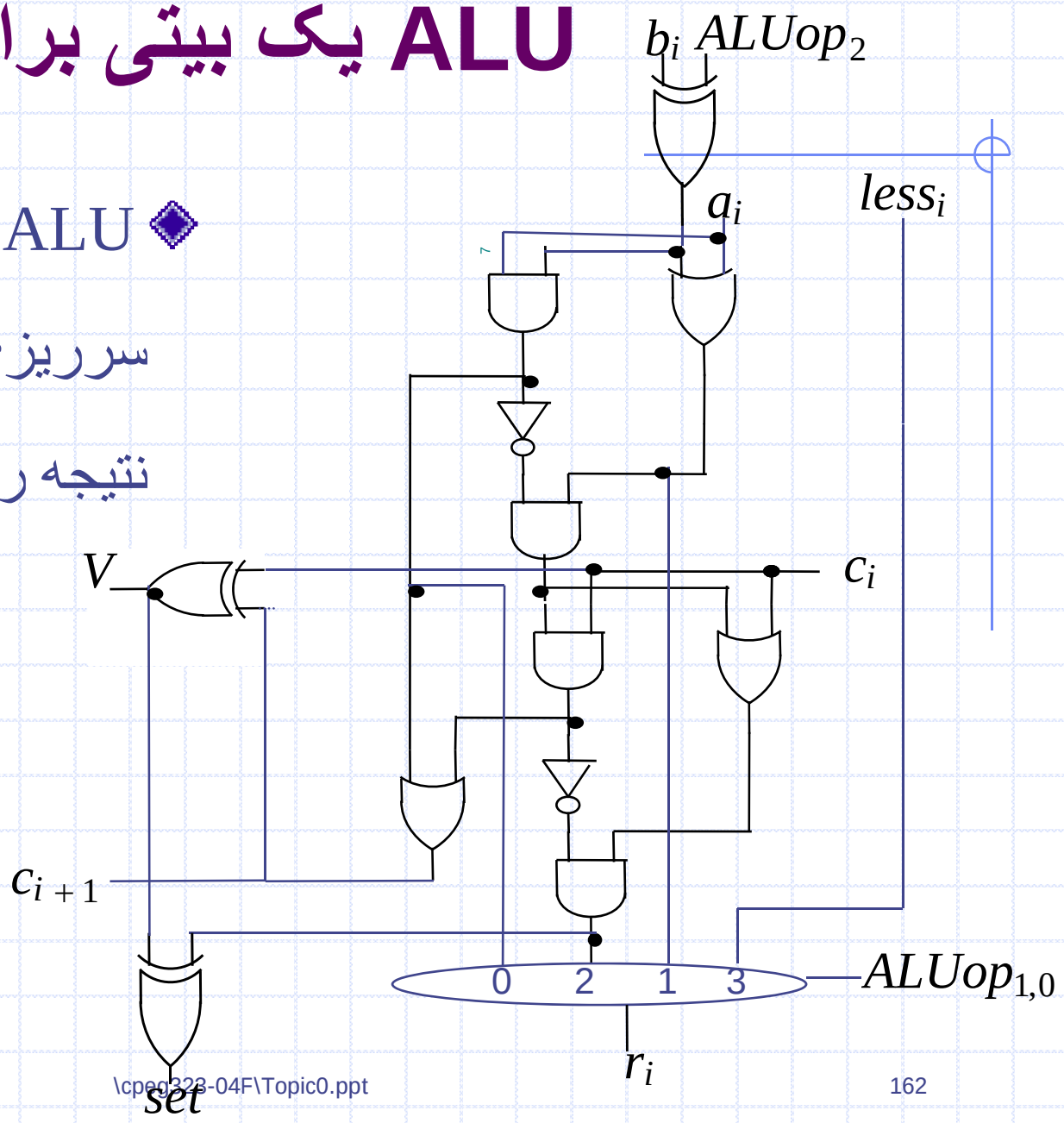


ALU یک بیتی برای MSB

- ALU برای MSB باید سرریزی را آشکار و علامت نتیجه را نیز نشان دهد.

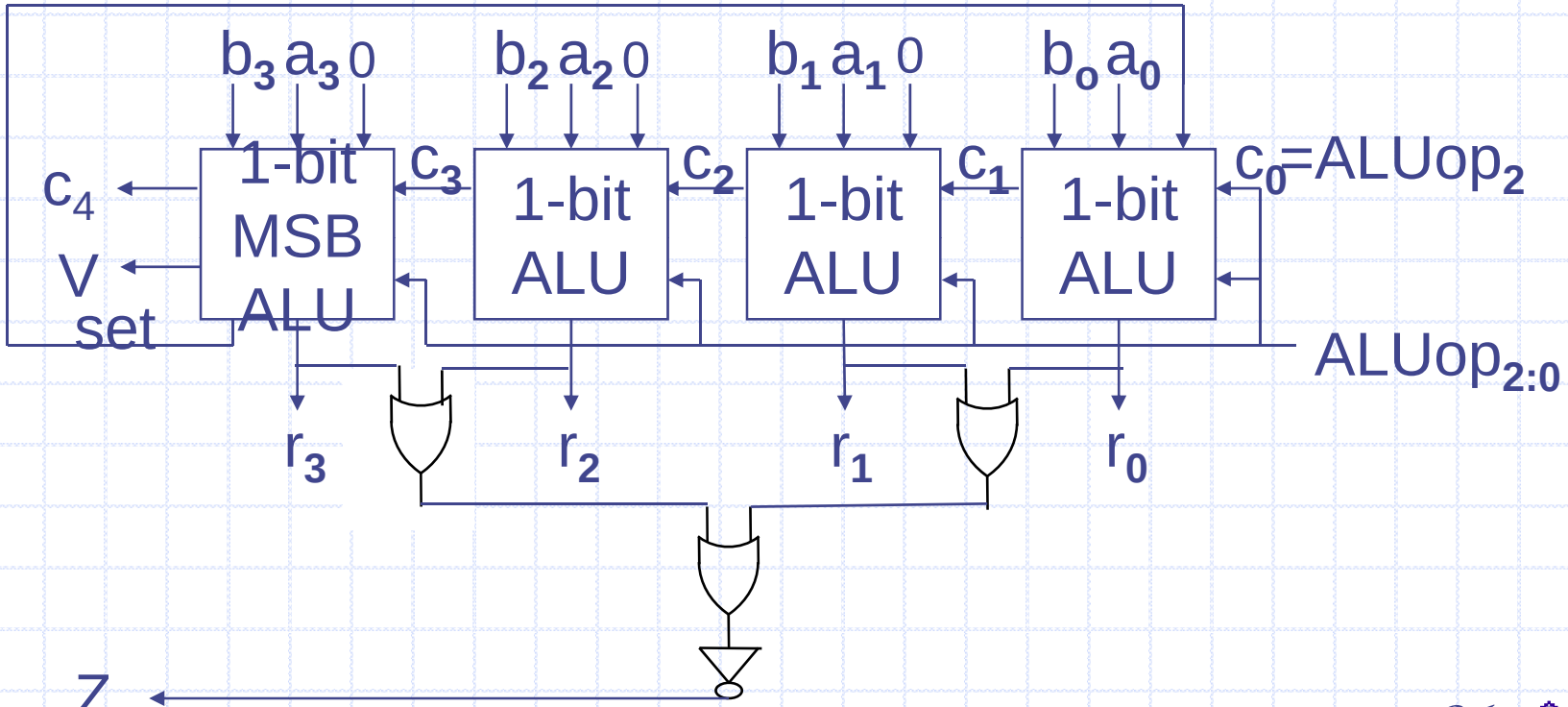
$$V = c_n \otimes c_{n-1}$$

$$set = (A < B)$$



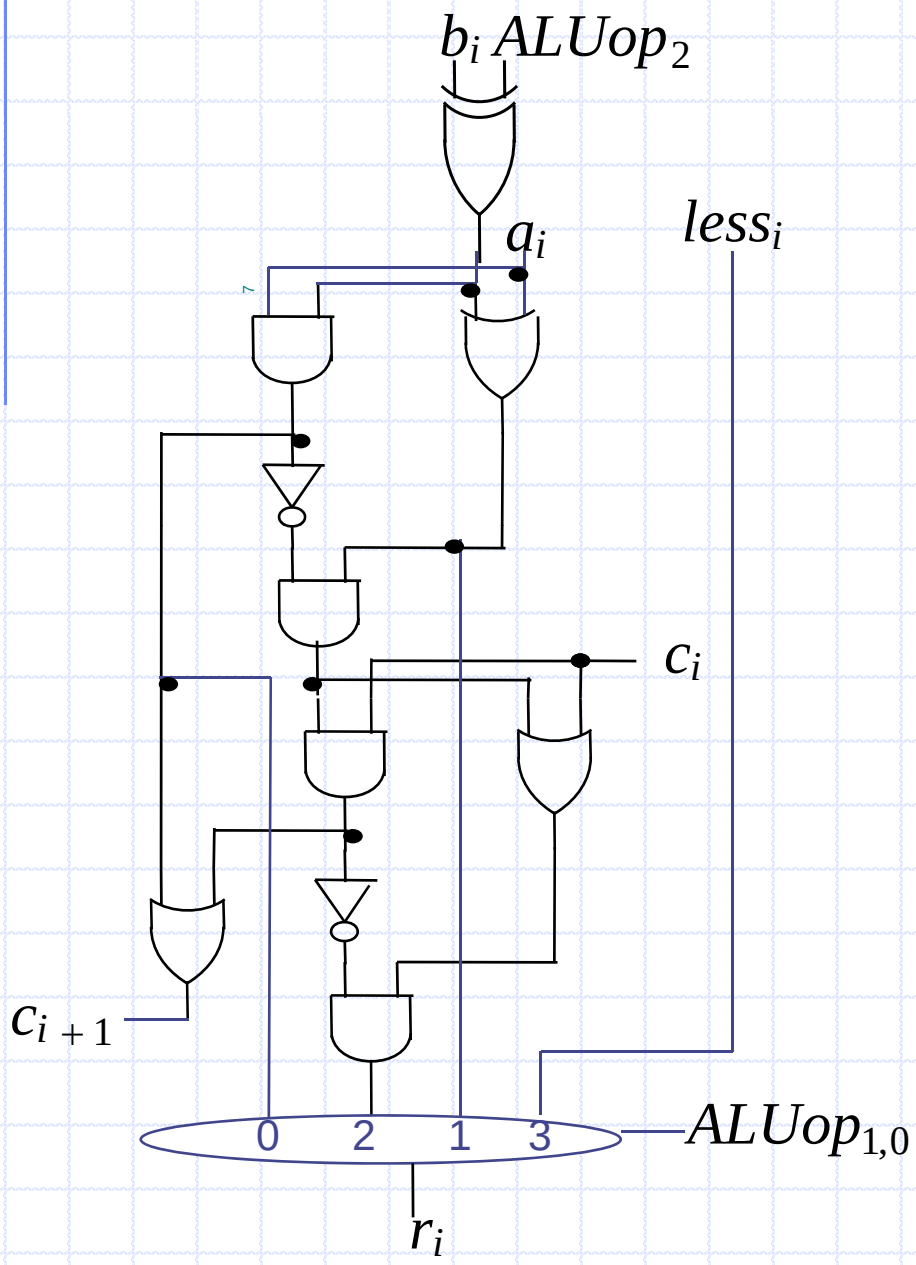
واحدهای محاسبه منطق(ALU)بزرگتر

- سه تا واحد محاسبه ومنطق یک بیتی و یک واحد محاسبه ومنطق یک بیتی MSBو 4 تا گیت NOR میتوانند باهم الحاق شوند و تشکیل یک واحد محاسبه و منطق (ALU) 4 بیتی را دهند.



- 31 عدد واحد محاسبه و منطق یک بیتی و یک واحد محاسبه ومنطق یک بیتی MSBو 32 تا گیت NORمی توانند باهم الحاق شوند و تشکیل یک واحد محاسبه و منطق (ALU) 32 بیتی را می دهند

محاسبه تعداد گیتها

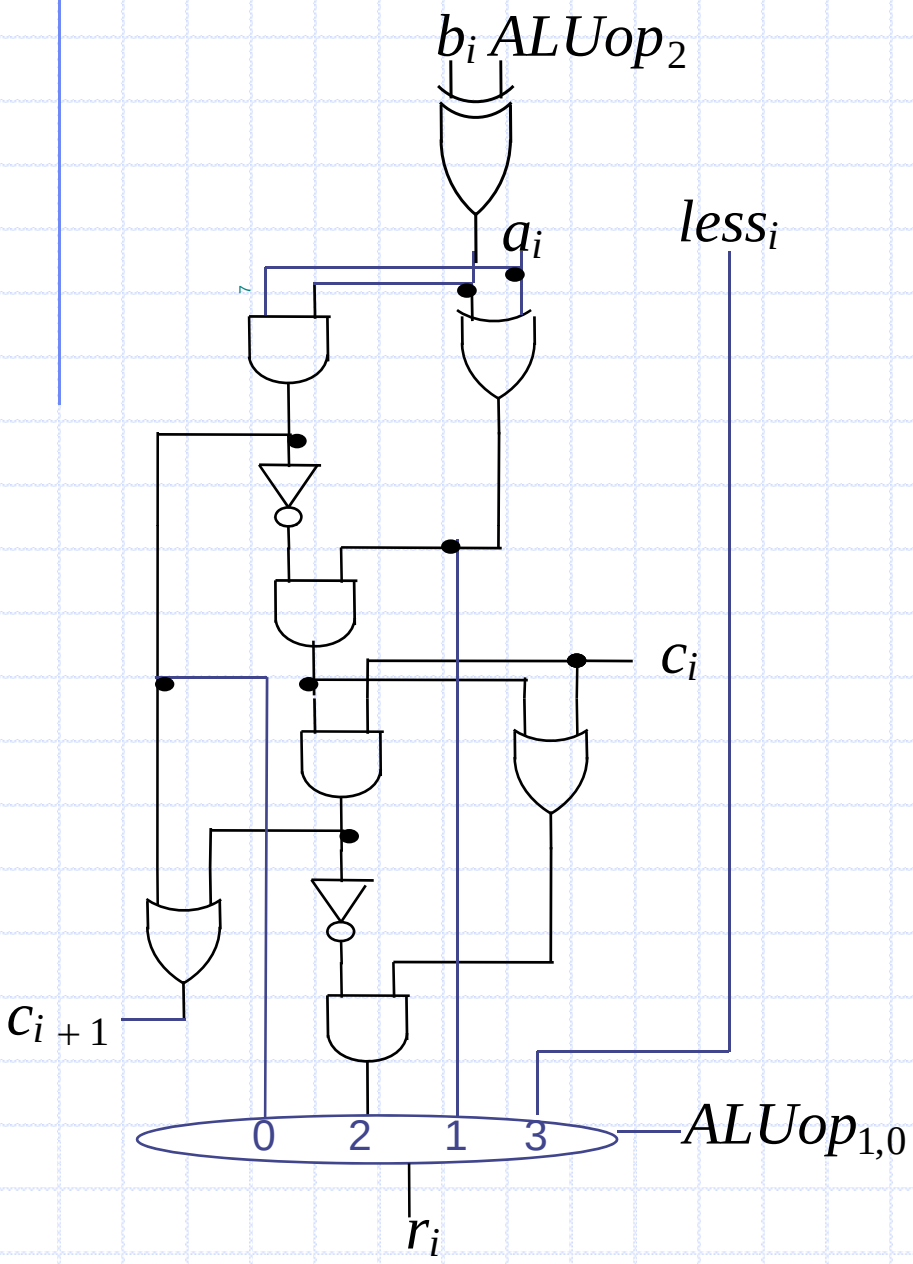


- فرض کنید
  - مالتی پالکسر 4 ورودی=5گیت
  - گیت XOR=3گیت
  - گیت AND/OR=1گیت
  - وارونه ساز=5/. گیت
- چند تا گیت نیاز است به وسیله
  - یک واحد محاسبه منطق 1 بیتی؟
  - یک واحد محاسبه منطق4 بیتی؟
  - یک واحد محاسبه منطق 32 بیتی؟
  - یک واحد محاسبه منطق n بیتی؟

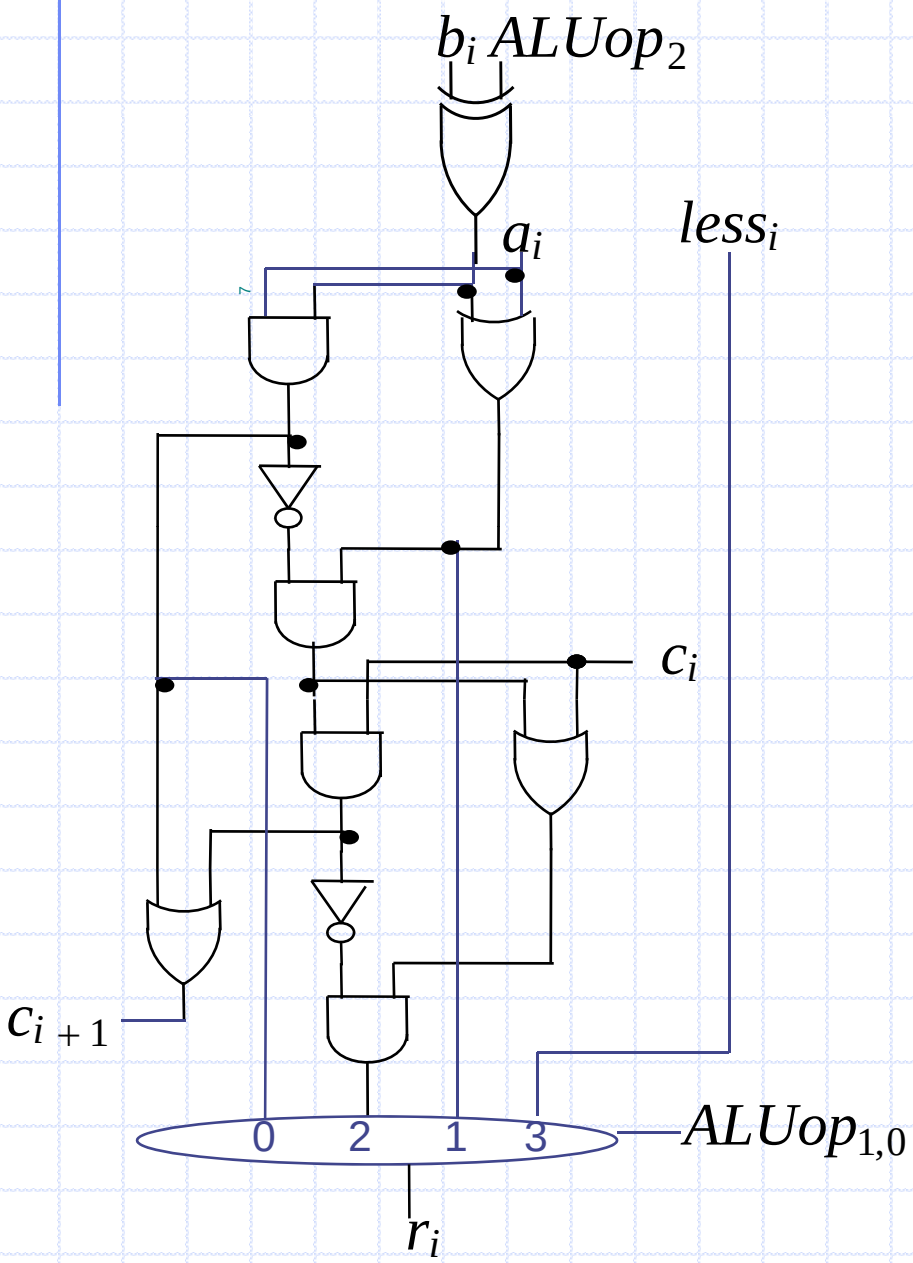
محاسبه تعداد گیتها

- فرض کنید
  - مالتی پالکسر 4 ورودی=5گیت
  - گیت XOR=3گیت
  - گیت AND/OR=1گیت
  - وارونه سا ز=5/. گیت
- چند تا گیت نیاز است به وسیله
  - یک واحد محاسبه منطق 1 بیتی؟16
  - یک واحد محاسبه منطق4 بیتی؟4*16
  - یک واحد محاسبه منطق 32 بیتی؟32*16
  - یک واحد محاسبه منطق n بیتی؟n*16

در یک واحد محاسبه و منطق n بیتی(n-1) عدد گیت ORبا 2 ورودی و یک وارونه ساز و یک گیتXOR



تاخیر گیت ها

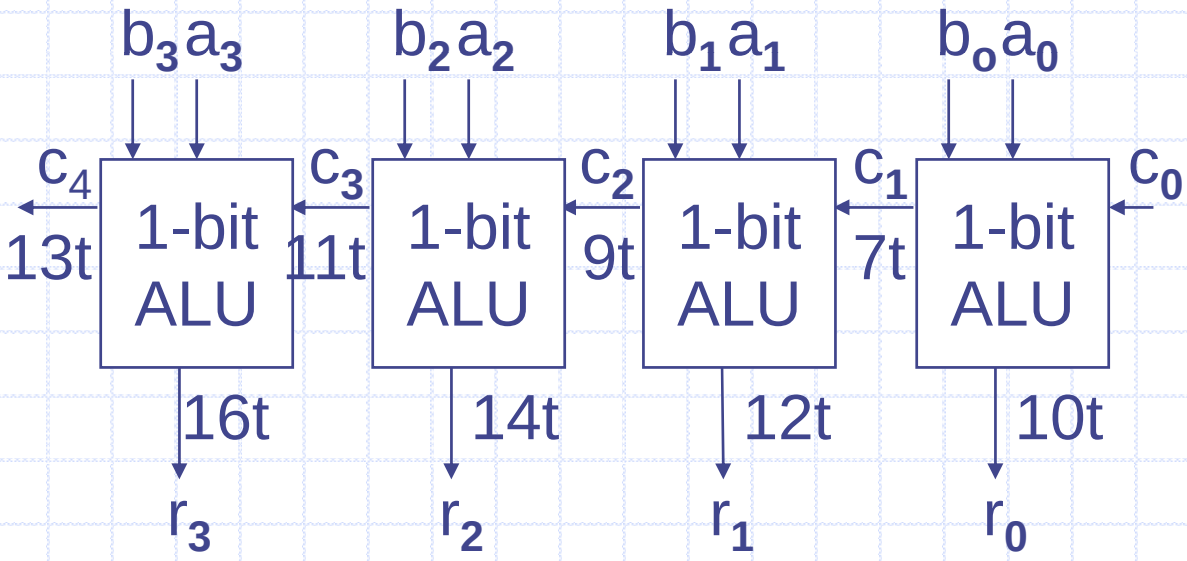


- فرض کنیدتاخیر های
  - مالتی پالکسر 4 ورودی=2t
  - گیت XOR=2t
  - گیت AND/OR=1t
  - وارونه سا ز=1t
- چقدر است تاخیر
  - یک واحد محاسبه منطق 1 بیتی؟10t
  - یک واحد محاسبه منطق4 بیتی؟
  - یک واحد محاسبه منطق 32 بیتی؟
  - یک واحد محاسبه منطق n بیتی؟

برای محاسبه zتاخیر اضافه نیاز است

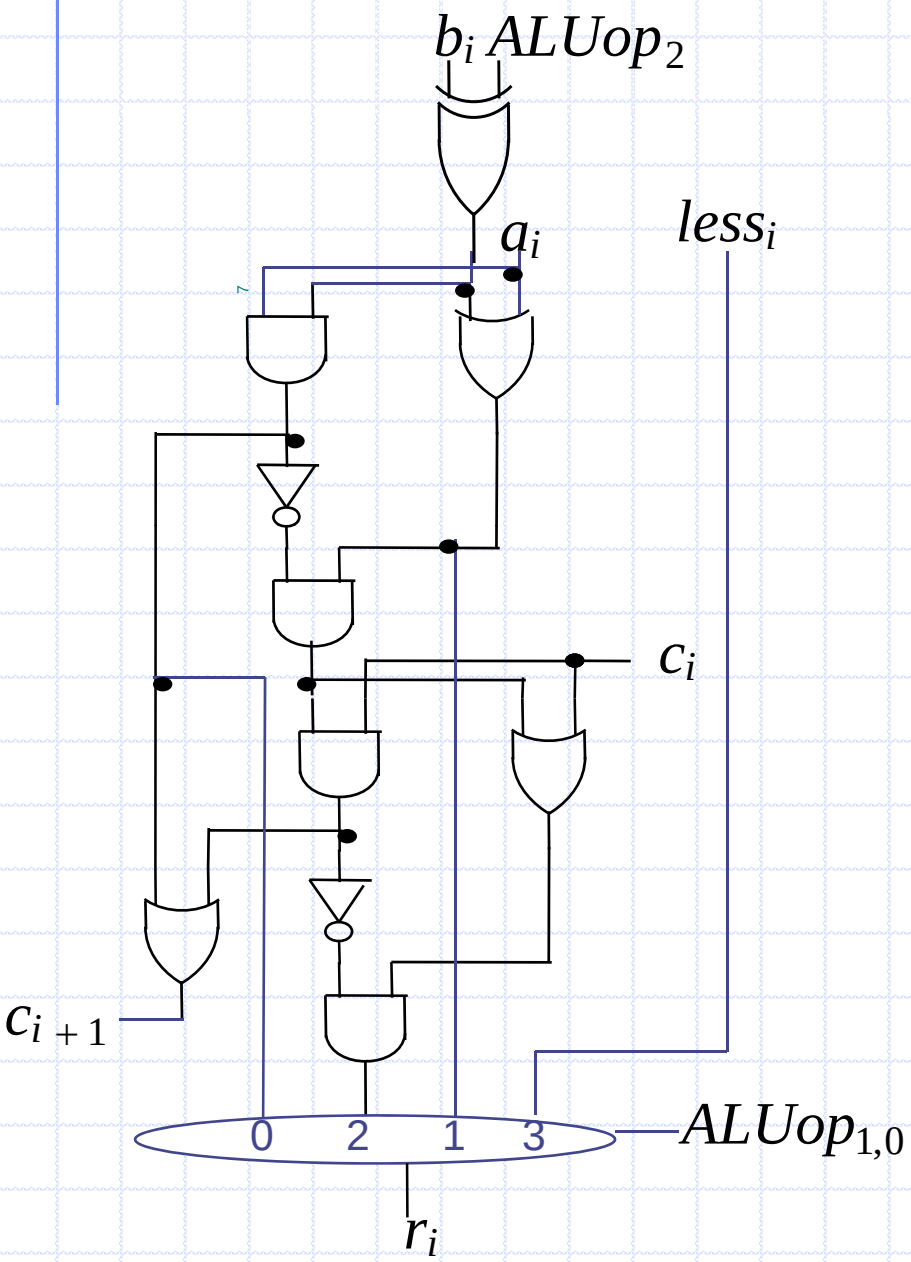
جمع کننده کری موج دار

- در طراحی قبلی کری"موج دار شده"از یک واحد محاسبه و منطق یک بیتی برای بعدی تولید می شود



- این ها موجب می شوند که تا اندازه ای طراحی کند شود
- Zدر لحظه 19tآماده است

تاخیر گیت

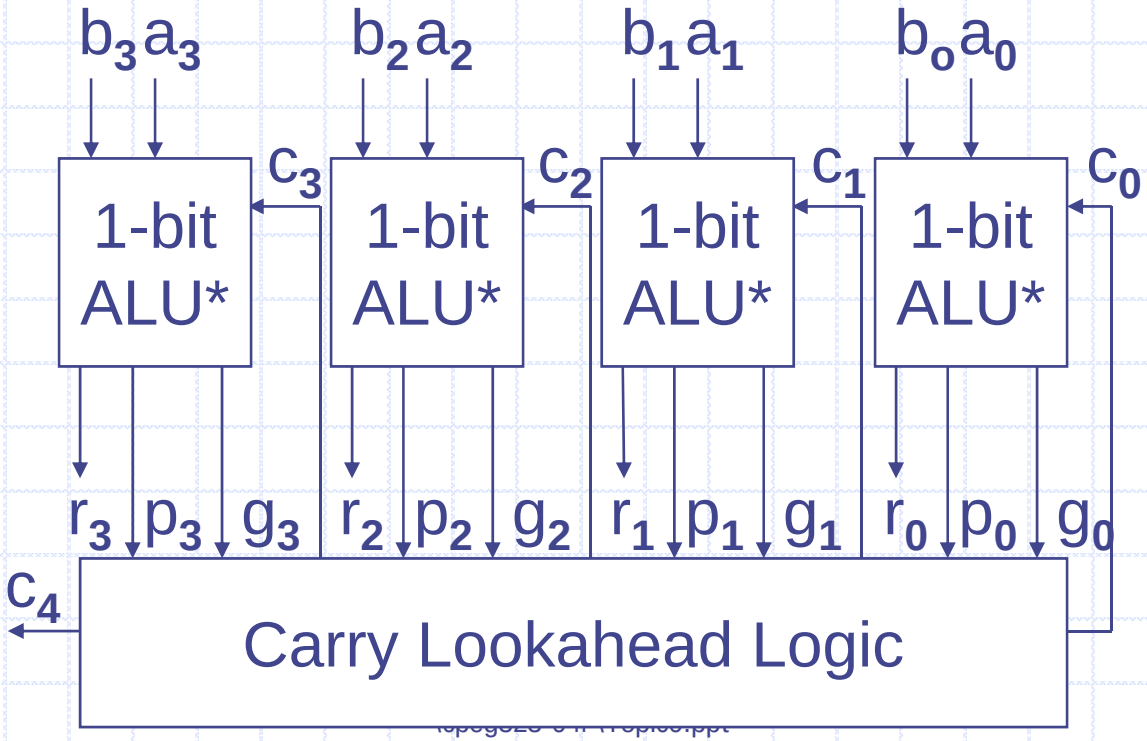


- فرض کنیدتاخیر های
  - مالتی پالکسر 4 ورودی=2t
  - گیت XOR=2t
  - گیت AND/OR=1t
  - وارونه سا ز=1t
- چقدر است تاخیر
  - یک واحد محاسبه منطق 1 بیتی؟10t
  - یک واحد محاسبه منطق4 بیتی؟16t
  - یک واحد محاسبه منطق 32 بیتی؟(2*32+8)t=72t
  - یک واحد محاسبه منطق n بیتی؟(2n+8)t

سطح OR دوورودی و 1 وارونه $\lceil \log_2(n) \rceil$
سازبرای محاسبه Zنیاز داریم

Carry Lookahead Adder (CLA)

- در یک CLA کری ها به صورت همزمان از یک carry lookahead logic(CLL) محاسبه می شوند



معادله منطقی کری

- معادله منطقی کری هست:

$$C_{i+1} = a_i b_i + (a_i + b_i) C_i$$

- ما یک سیگنال مشابه تعریف می کنیم propagate

$$p_i = a_i + b_i$$

- و یک سیگنال تولید میشود generate

$$g_i = a_i b_i$$

- این امکان را ایجاد می کند که معادله منطقی کری دوباره به صورت زیر نوشته شود

$$C_{i+1} = g_i + p_i c_i$$

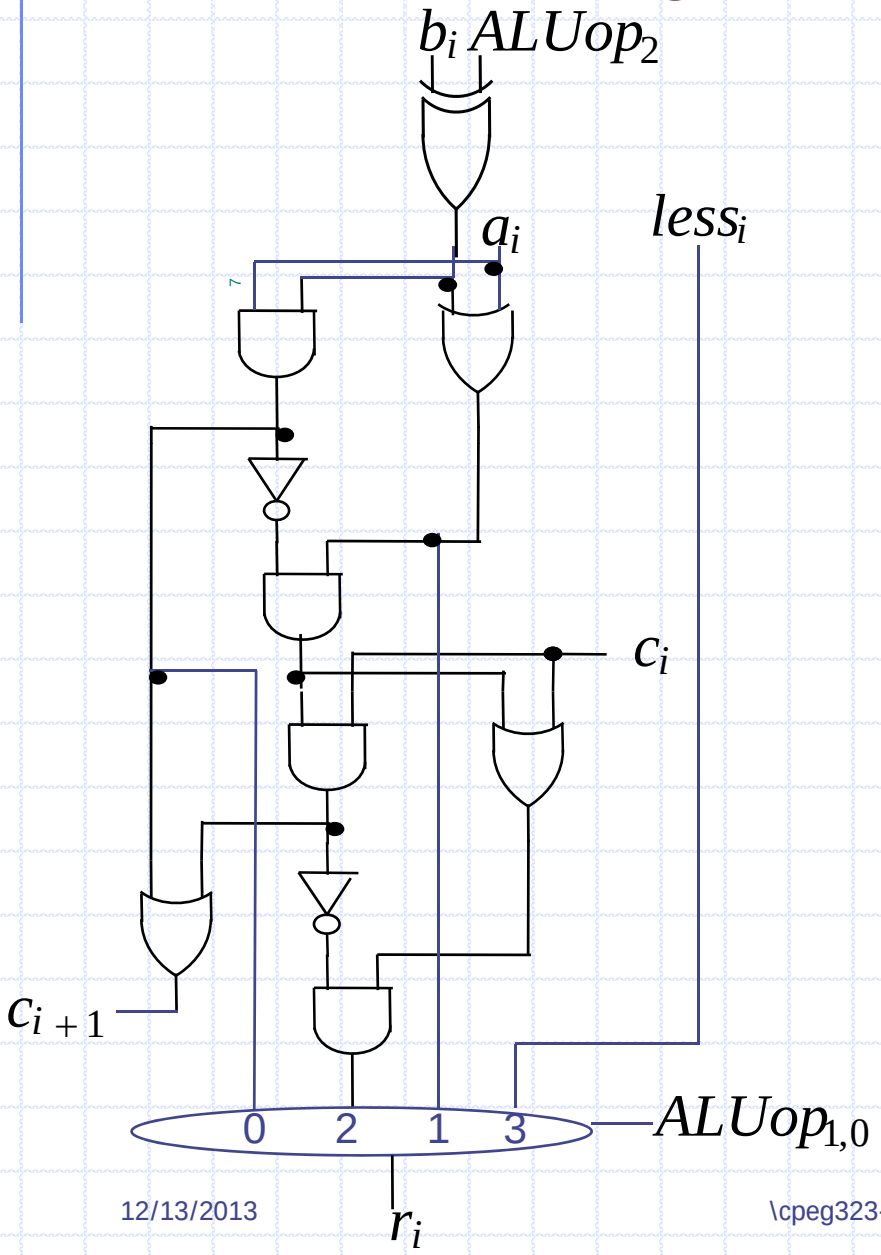
Carry Lookahead Logic

- برای یک 4 carry lookahead adder بیتی کری ها به صورت زیر محاسبه می شوند

$$c1 = g0 + p0c0$$
$$c2 = g1 + p1c1 = g1 + p1(g0 + p0c0)$$
$$= g1 + p1g0 + p1p0c0$$
$$c3 = g2 + p2c2 = g2 + p2(g1 + p1g0 + p1p0c0)$$
$$= g2 + p2g1 + p2p1g0 + p2p1p0c0$$
$$c4 = g3 + p3c3 = g3 + p3(g2 + p2g1 + p2p1g0 + p2p1p0c0)$$
$$= g3 + p3g2 + p3p2g1 + p3p2p1g0 + p3p2p1p0c0$$

- چند تا گیت CLL 4بیتی نیاز داریم اگرکه گیت ها بتوانند گنجایش ورودی نا محدود داشته باشند؟
- اگر هرسطح منطقی تنها تاخیر 1t داشته باشد CLL دارای 2t تاخیر است در نتیجه این تمرین ممکن است واقع بینانه نباشد.

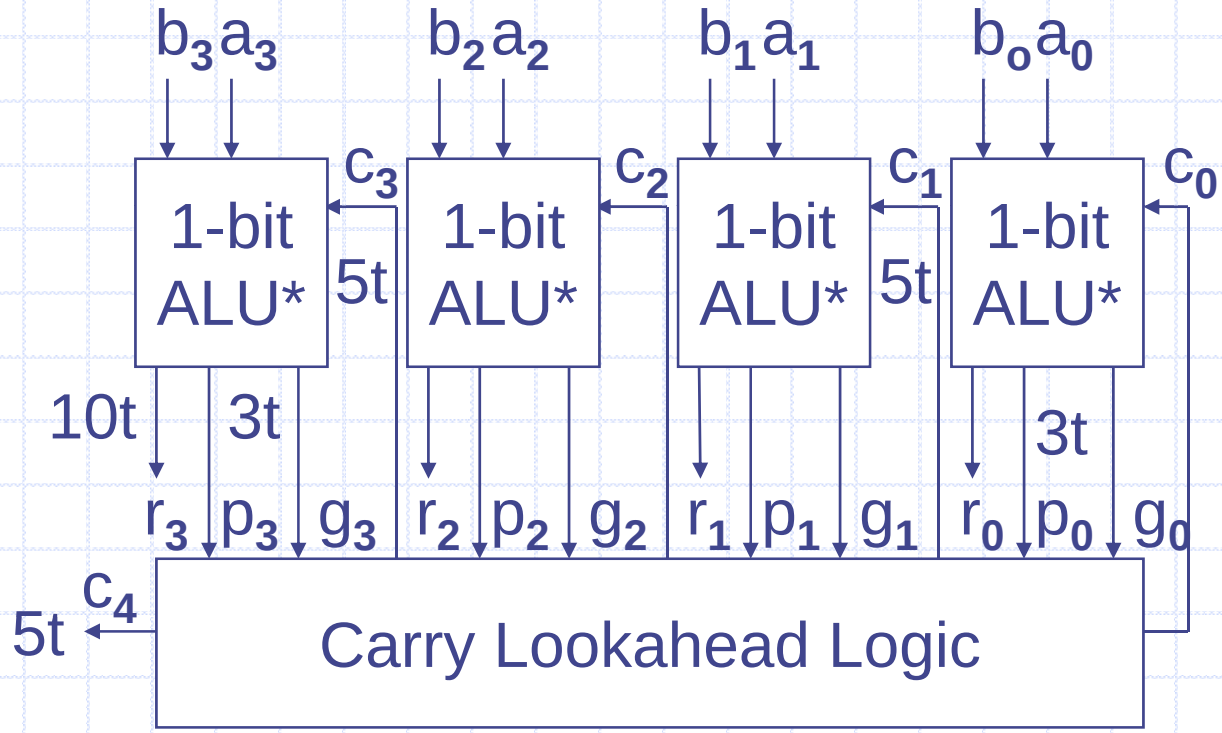
تعدیل کردن واحد محاسبه ومنطق یک بیتی



- چه طور ما می توانیم یک واحد محاسبه و منطق یک بیتی را تعدیل کنیم اگر که در CLAمورد استفاده باشد؟
- چند تا گیت تعدیل شده برای واحد محاسبه و منطق یک بیتی نیازاست؟
- چند تا گیت برای CLA 4بیتی نیاز است؟
- چقدر تاخیر داریم تا زمانی که PiوGiآماده شوند؟

تعیین زمان 4 CLA بیتی

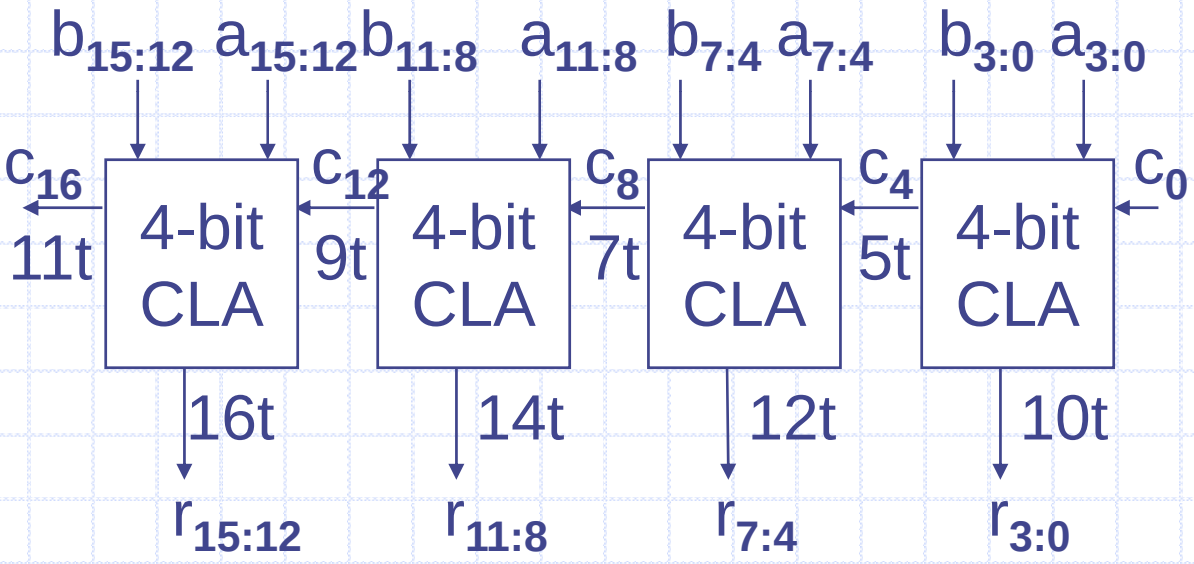
- در یک Carry lookahead adder کری ها به صورت همزمان محاسبه می شوند از Carry lookahead logic



- این طراحی به 15*4+14=74 گیت احتیاج دارد بدون محاسبه zیاv

واحد محاسبه و منطق 16بیتی .ورژن1

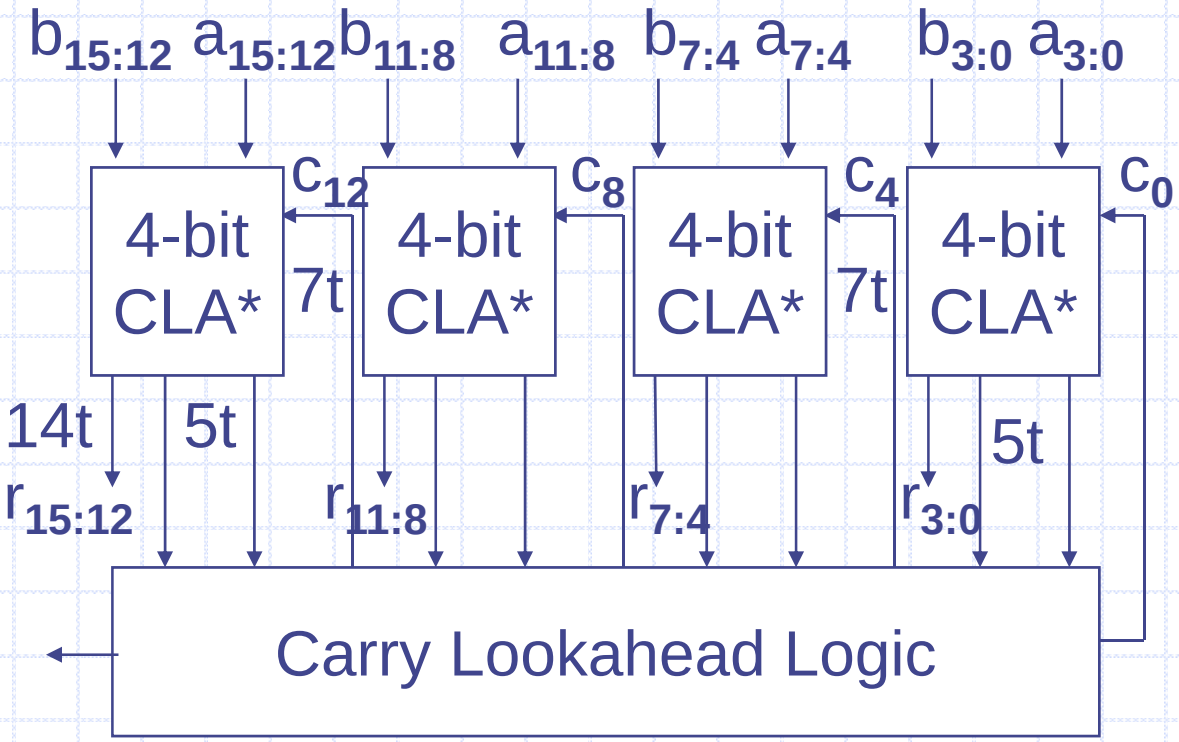
- یک ALU 16بیتی ساخته شده از الحاق کردن 4 تا CLA 4بیتی و اجازه انتقال کری "موج دار "بین "بلاک های" 4بیتی



- این طراحی نیاز دارد به 74*4=296گیت بدون محاسبه V یا Z

واحد محاسبه و منطق 16بیتی . ورژن2

- پیشنهاد دیگر استفاده از دومین سطح Carry lookahead logicاست.
- این پیشنهاد سریعتر است اما به گیت های بیشتری نیاز دارد 16*15+5*14=310تا گیت.



CLA 4بیتی

- CLA 4بیتی(BLOCK CLA) شبیه اولین CLA 4بیتی است به جز محاسبات CLLیک "تولید بلاک" و "منتشر کردن بلاک" به جای خروجی کری
- بدین ترتیب محاسبه

c4 = g3 + p3g2 + p3p2g1 + p3p2p1g0 + p3p2p1p0c0

- جایگزین شده به وسیله

P3:0 = p3p2p1p0

G3:0 = g3 + p3g2 + p3p2g1 + p3p2p1g0

Note: c4 = G3:0 + P3:0c0

- این پیشنهاد محدود می کند حداکثرFan را به 4 و CLL هنوز نیاز به 14 گیت دارد

نتایج

- یکALU nبیتی می تواند به وسیله الحاق کردن nتا ALU یک بیتی طراحی شود . CLLمی تواند برای افزایش سرعت محاسبات استفاده شود.
- انوع گوناگون گزینش های طراحی برای طراحی ALUوجود دارد
- بهترین طراحی بستگی دارد به ناحیه تاخیر و توان نیازمندی ها که براساس تنوع تکنولوژی اصولی است

ضرب وتقسيم صحيح

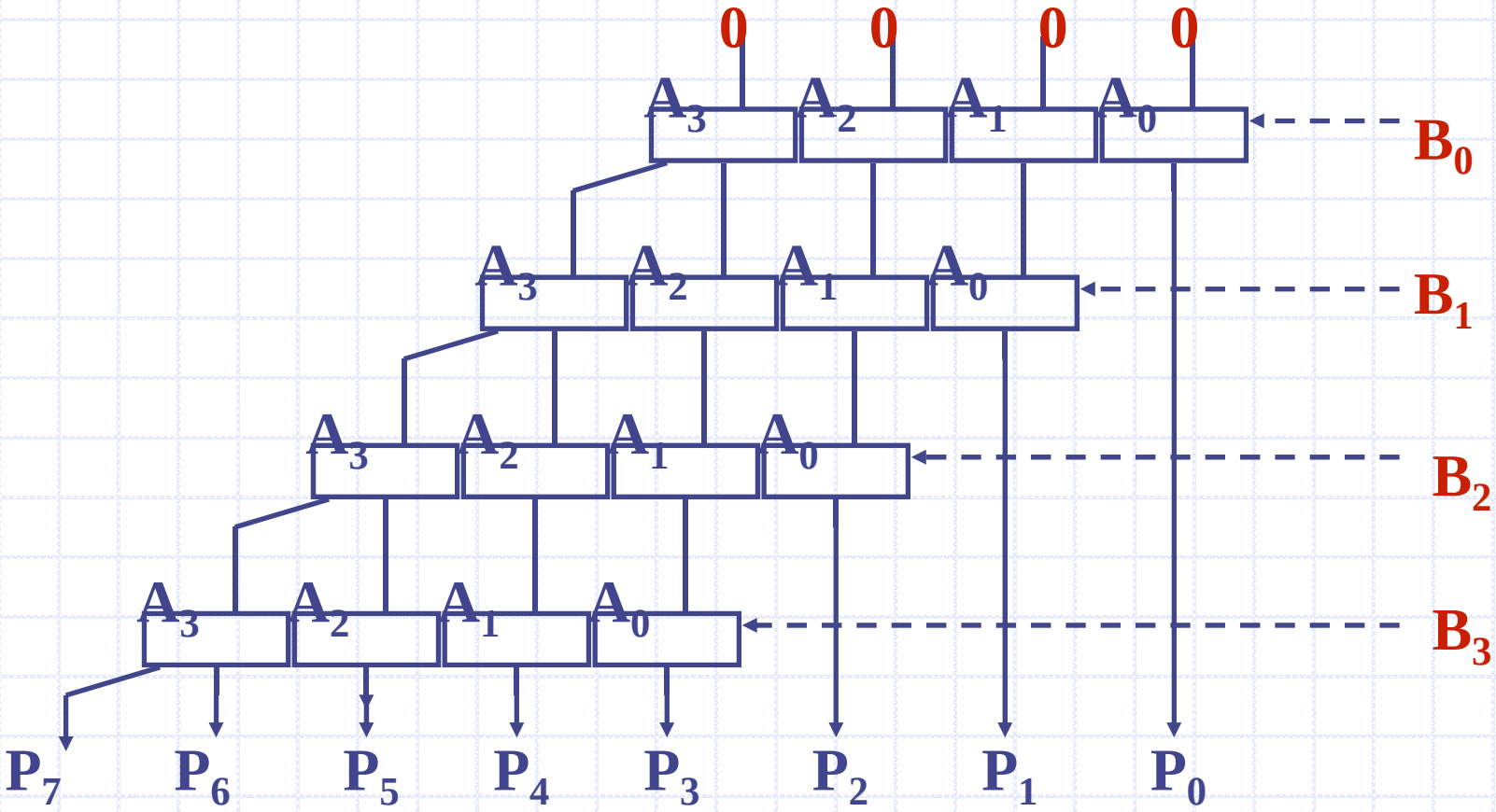
ضرب صحيح بدون علامت

- مثال مداد و كاغذ

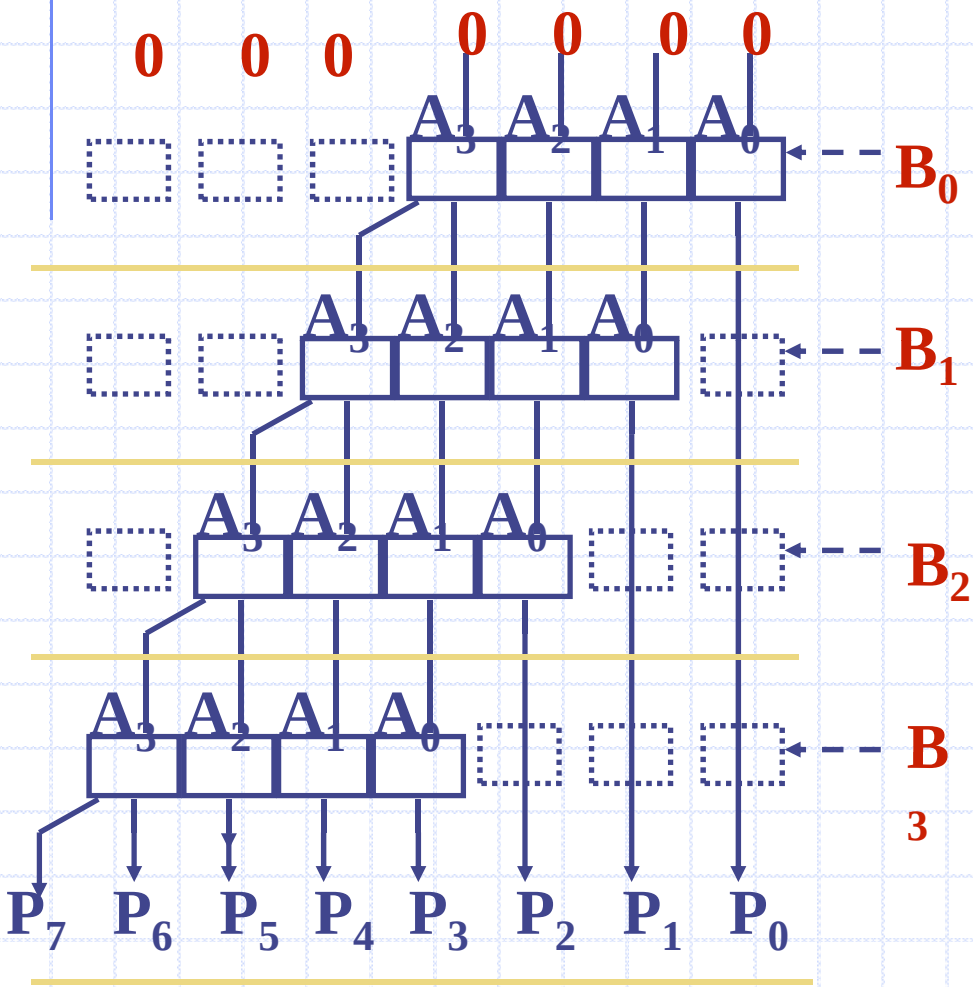
مضروب	1000
مضروب فيه	* 1001
	1000
	0000
	0000
	1000
حاصلضرب	01001000

مشاهدات

- mبیت*nبیت=m+n بیت حاصلضرب
- انباشته های i مرحله $A * 2^i$ if $B_i == 1$



این چگونه کار می کند؟



- در هر مرحله 2 بار به چپ شیفت میدهد.
- استفاده از بیت بعدی Bبرای اینکه تعیین کند که آیا با مضروب شیفت داده شده جمع شود یا نه.
- جمع کردن 2nبیت حاصل ناتمام در هر مرحله

محاسبات ساده ریاضی

A*B

اگر B N تا بیت داشته باشد

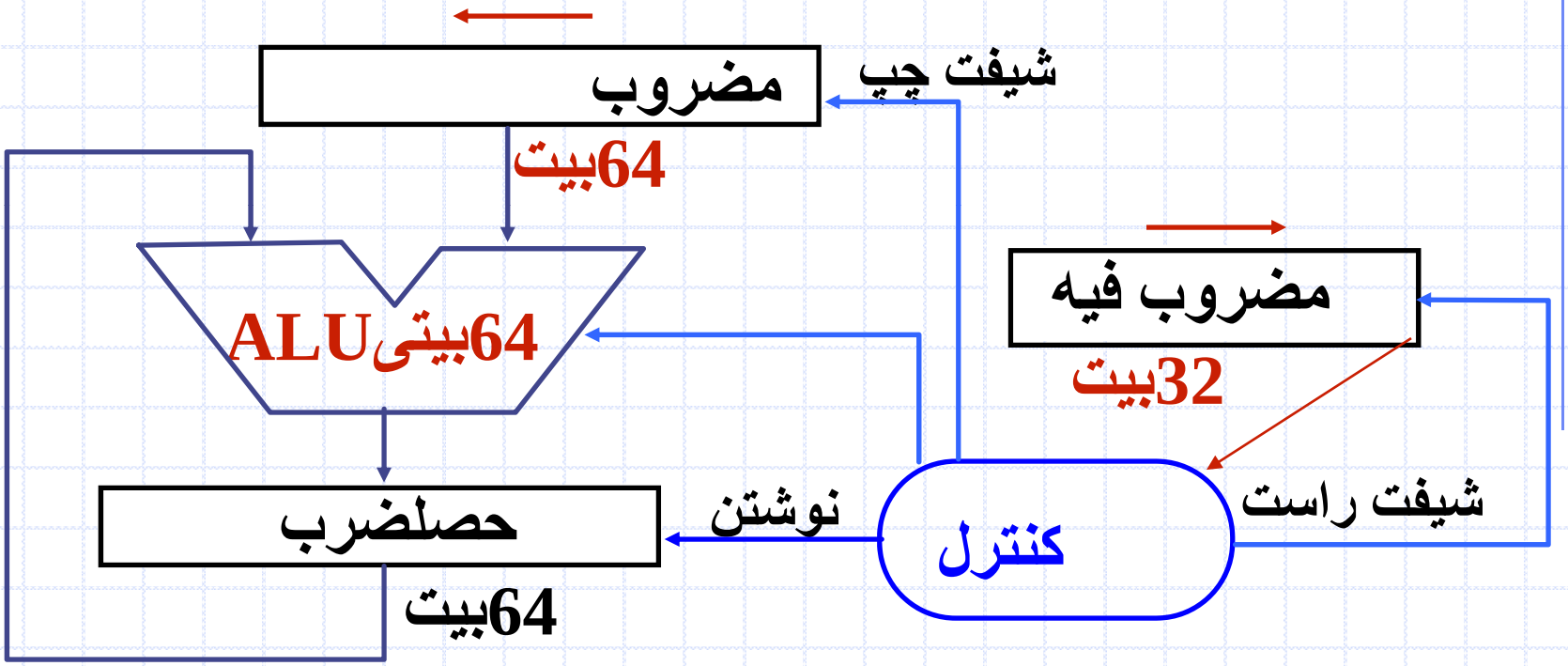
$$B = \sum_{i=0}^{n-1} (2^i * B_i) \qquad Product_n = \sum_{i=0}^{n-1} (2^i * B_i * A)$$

سپس زمانی که B N+1 بیت دارد

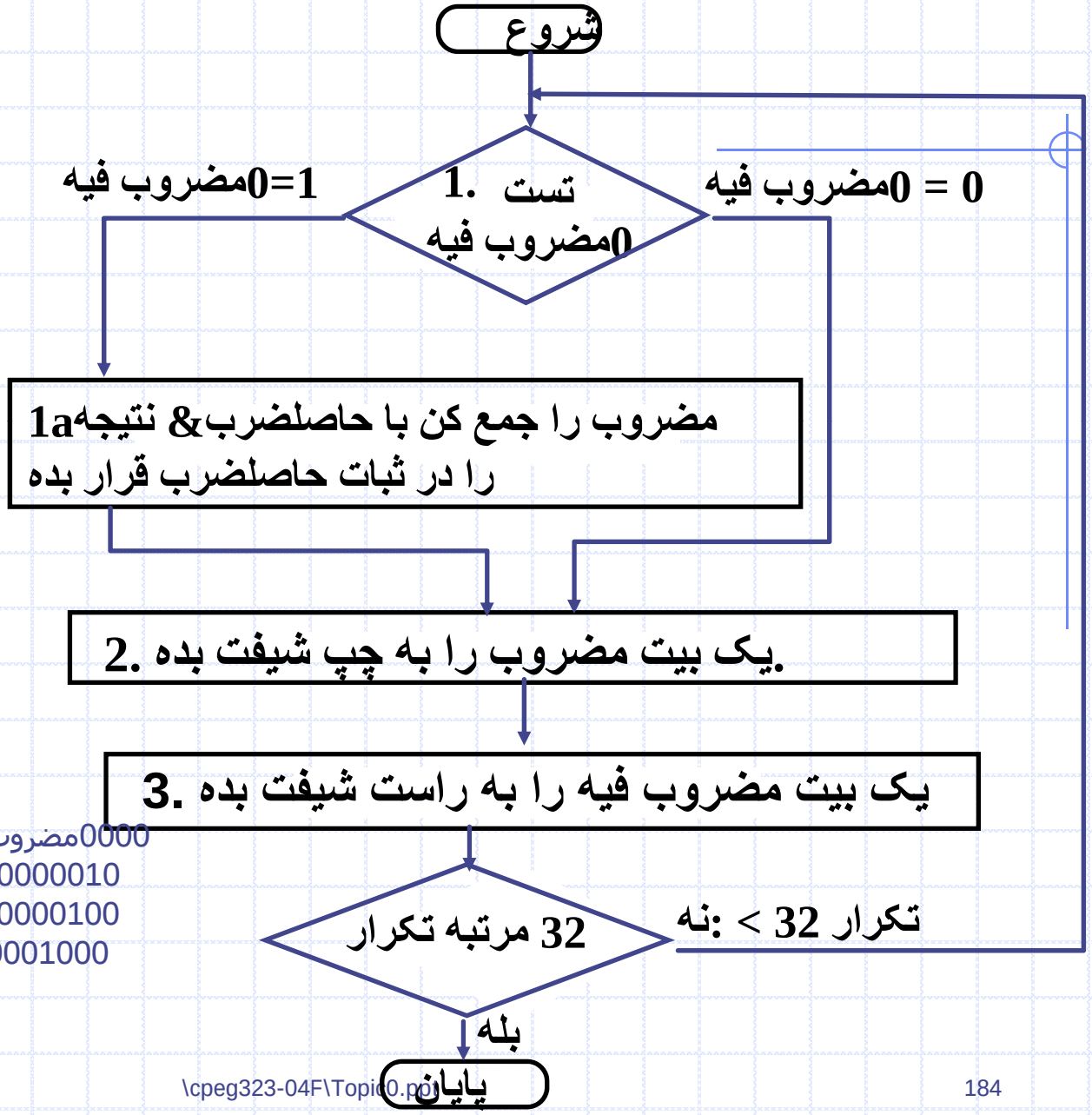
$$Product_{n+1} = \sum_{i=0}^{n} (2^i * B_i * A) = 2^n * B_n * A + Product_n$$

- Aرا یک مرتبه به چپ شیفت دادن :2^i*A .
- این واضح است که ضرب مشتمل بر شیفت و جمع تکراری است.

سخت افزار ضرب ورژن 1



الگوریتم ضرب ورژن 1

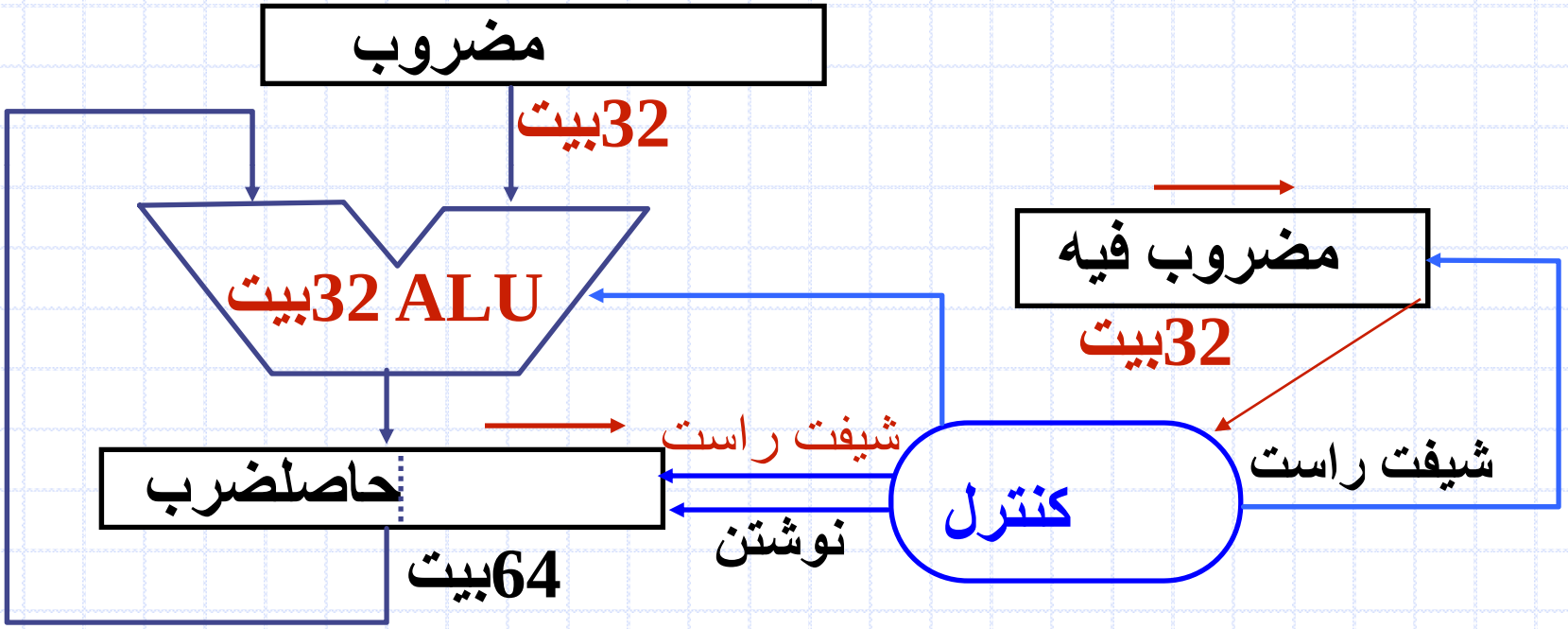


حاصلضرب	مضروب فیه	0000مضروب
0000	0011	00000010
00000010	0001	00000100
00000110	0000	00001000
00000110		

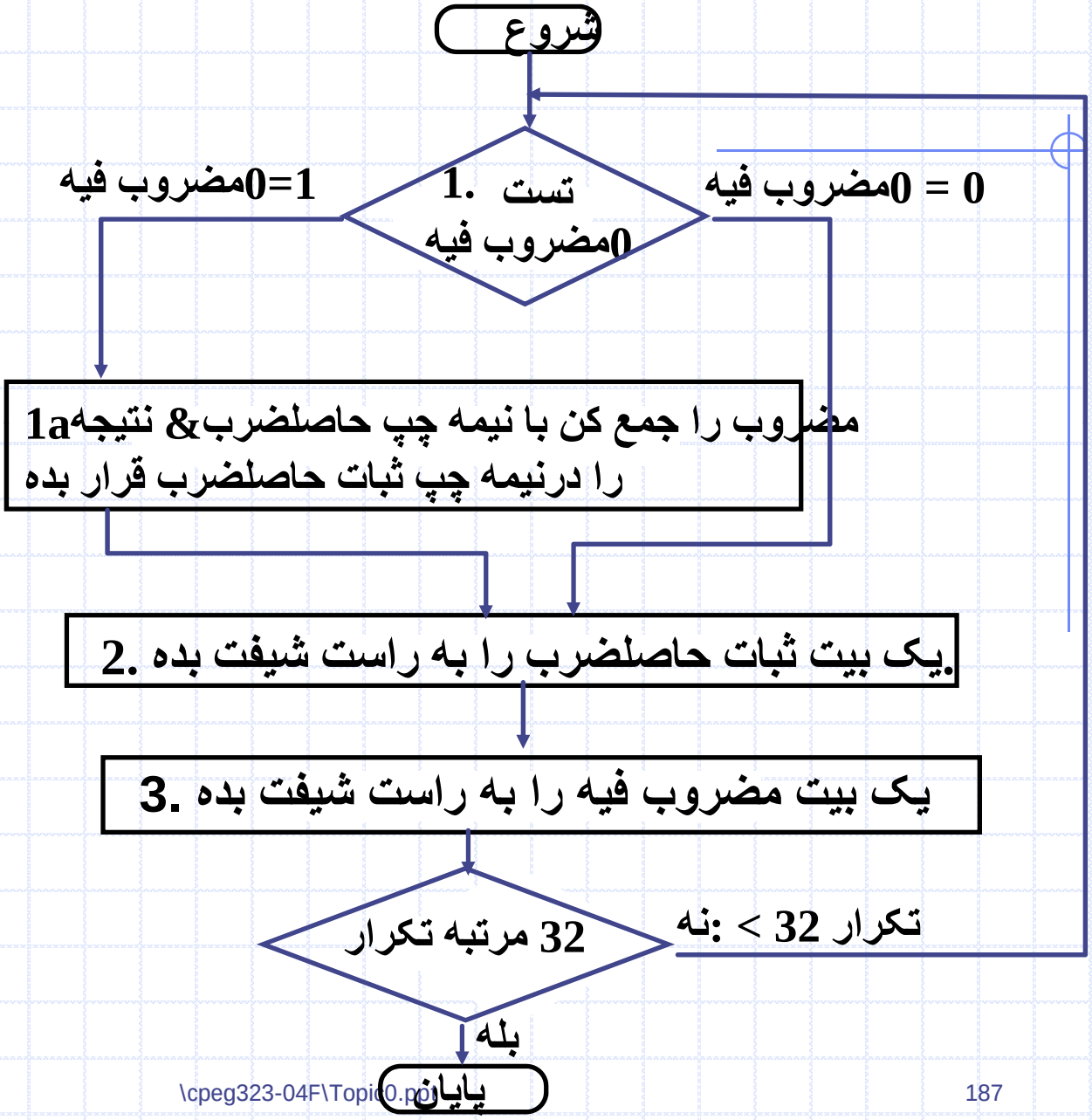
نظارت بر روی ورژن1ضرب

- نصف بیتها در مضروب همیشه صفر است =>64 بیت جمع کننده هدر رفته است
- مقدار دادن در سمت چپ مضروب به عنوان شیفت دادن=> کم ارزش ترین بیت حاصلضرب هرگز تغییر نمی کند
- به جای اینکه مضروب را به چپشیفت دهیم. حاصل را به راست شیفت می دهیم؟

سخت افزار ضرب ورژن2



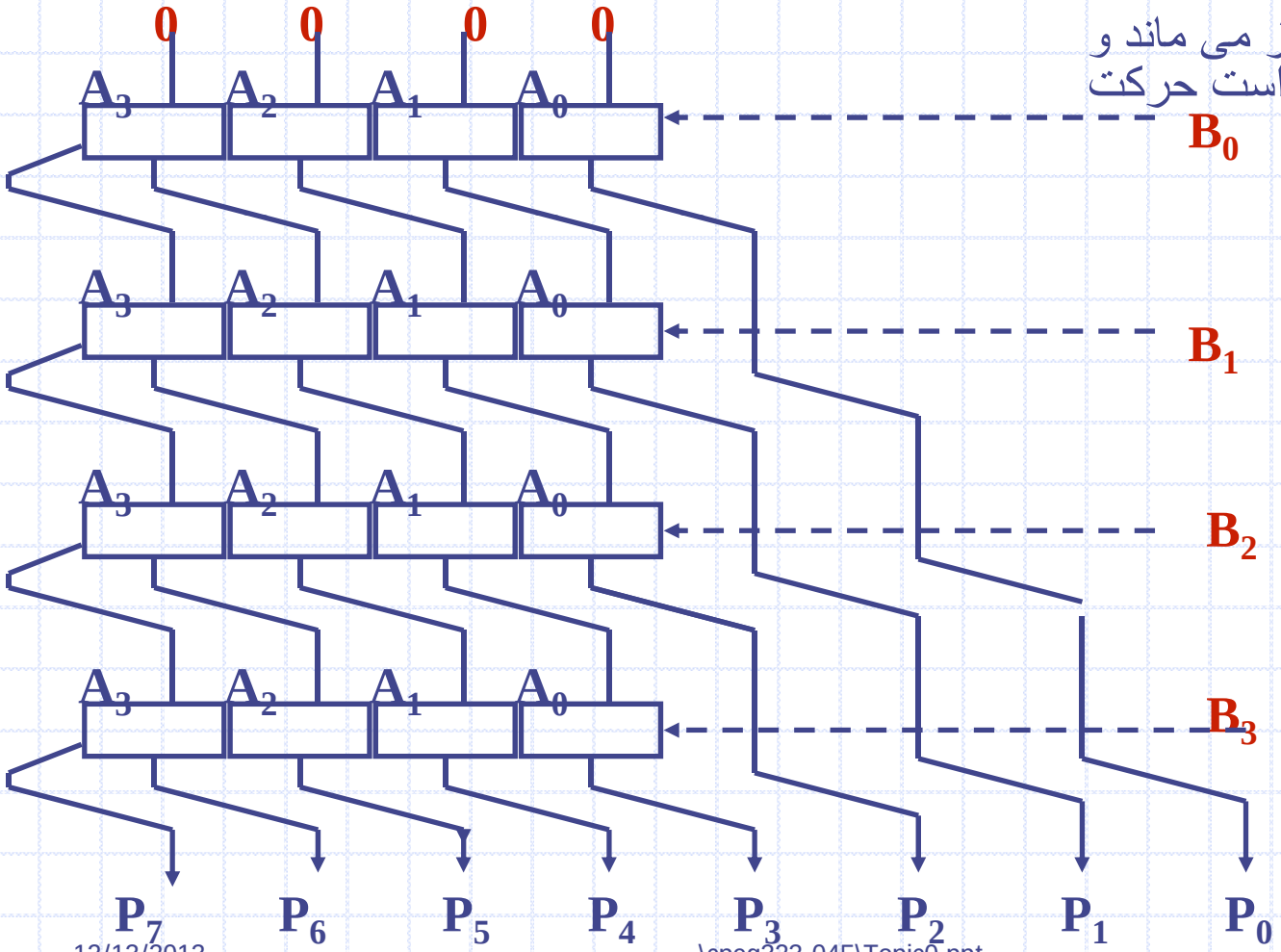
الگوریتم ضرب ورژن2



	حاصلضرب	مضروب فيه	مضروب
	0000 0000	0011	0010
1:	0010 0000	0011	0010
2:	0001 0000	0011	0010
3:	0001 0000	0001	0010
1:	0011 0000	0001	0010
2:	0001 1000	0001	0010
3:	0001 1000	0000	0010
1:	0001 1000	0000	0010
2:	0000 1100	0000	0010
3:	0000 1100	0000	0010
1:	0000 1100	0000	0010
2:	0000 0110	0000	0010
3:	0000 0110	0000	0010
	0000 0110	0000	0010

چه چیزی روی می دهد؟

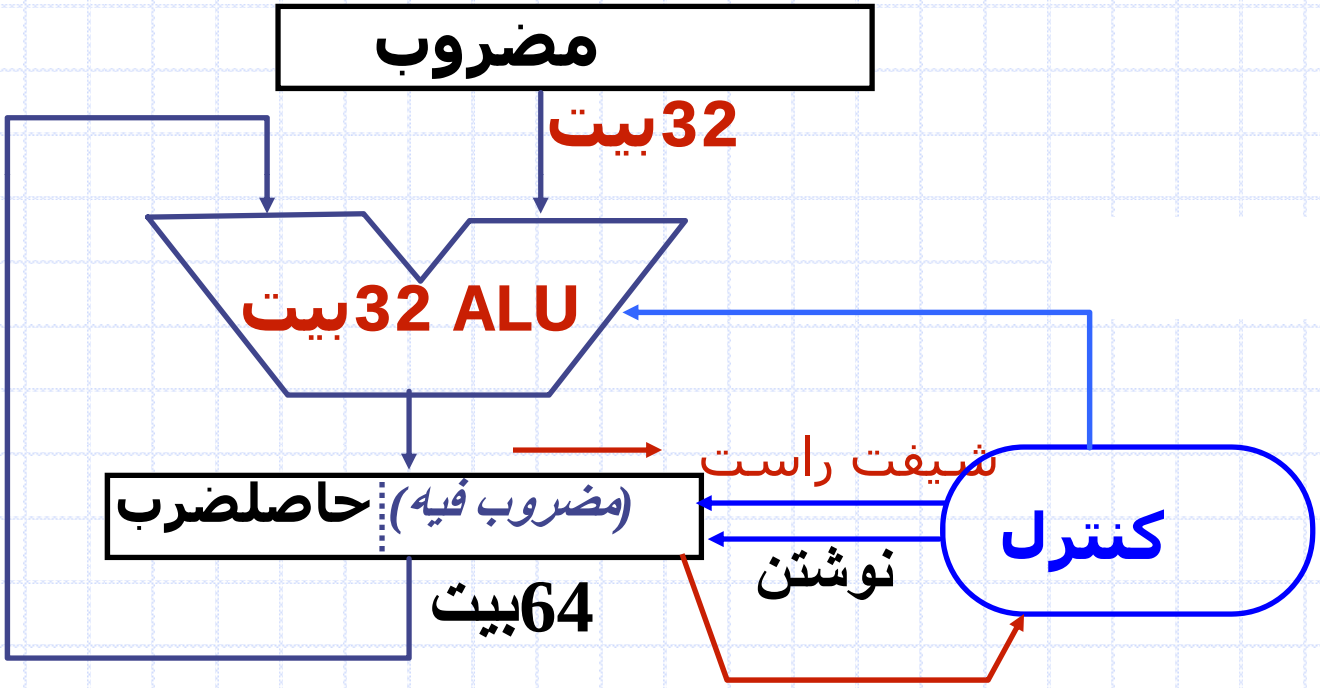
- مضروب فیه هنوز می ماند و حاصلضرب به راست حرکت می کند



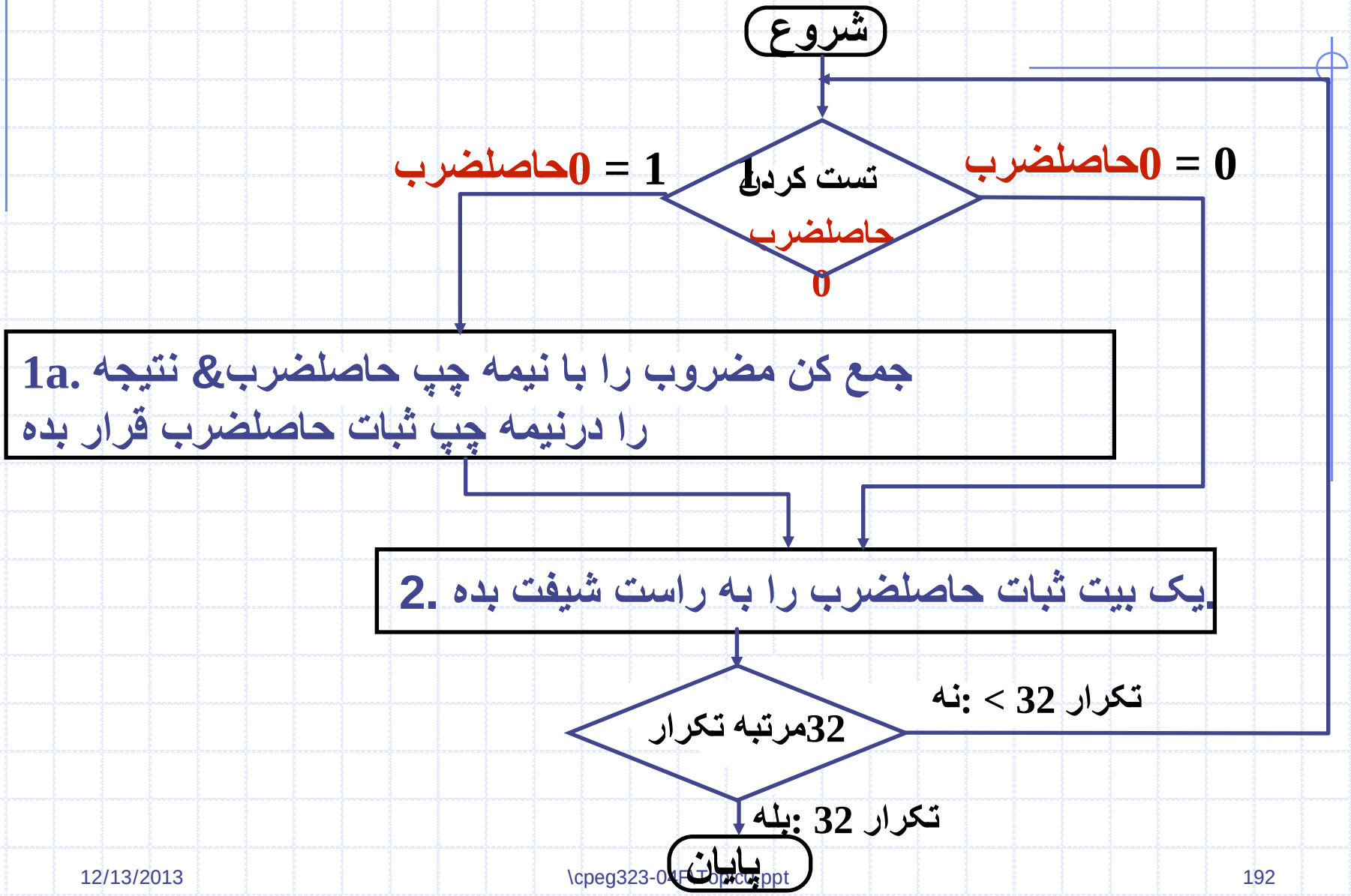
مشاهدات ورژن2ضرب

- فضای هدر رفته ثبات حاصلضرب که دقیقا هم اندازه مضرئب فیه است =>به هم پیوستن ثبات مضروب فیه و ثبات حاصلضرب

سخت افزار ضرب ورژن2



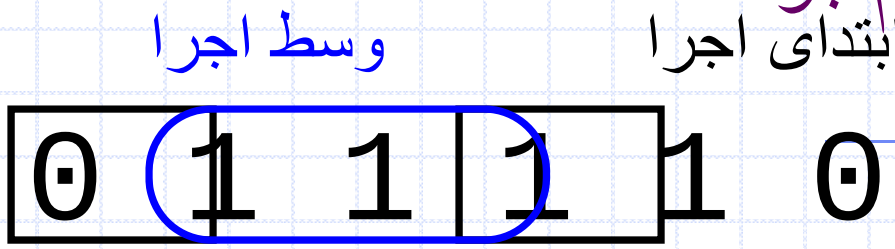
الگوریتم ضرب ورژن3



مشاهدات ورژن3ضرب

- دو مرحله در هر بیت است برای اینکه مضروب فیه & حاصلضرب به هم می پیوندد.
- ثبات های Lo,Hi MIPSهستند نیمه چپ و راست حاصلضرب
- MIPSدستور العمل ضرب بدون علامت را به ما می دهد
- درمورد ضرب علامت دار چی؟
  - آسانترین راه حل زمانی است که هردو مثبت باشند و به یاد داشته باشید داده تکمیل شده اجرا می شود(بیت علامت را دور بریز اجرا برای 31مرحله)
- کاربرد تعریف متمم 2
  - نیاز به گسترش علامت داده های بخصوص و طبقه بندی آنها در انتها
- الگوریتم بوت راه فوق العاده برای ضرب اعداد علامت دار است و از سخت افزار یکسان مانند قبلی استفاده میکند و چرخه ها را ذخیره می کند
  - می تواند تعدیل شود بهto handle multiple bits at a time
  -

الگوریتم بوت



- (بیت راست بیت جاری)

OP	مثال	توضیح	بیت جاری	بیت راست
sub	0001111000	شروع اجرا از 1S	1	0
none	0001111000	وسط اجرا از 1S	1	1
add	0001111000	پایان اجرای1s	0	1
none	0001111000	وسط اجرا از 1s	0	0

- ابتکاری برای سرعت (موقعی که شیفت سریعتر از addبود). یک رشته از 1sها را در مضروب فیه با نخستین تفریق جایگزین کنید

```
  -1
+ 10000
-------
  01111
```

- ما ابتدا یک 1 می بینیم و سپس در جمع بعدی ////////////////////////////

مثال بوت(2*7)

عملگر	مضروب	حاصل	بعدی؟
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+1110 1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+0010 0001 1100 1	shift
4b.	0010	0000 1110 0	done

مثال بوت(3-*2)

عملگر	مضروب	حاصل	بعدی ?
0. initial value	**0010**	**0000 1101 0**	**10 -> sub**
1a. P = P - m	1110	+1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+ 0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+1110	
3a.	0010	1110	
1011 0	shift P		
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

ضرب علامت دار

- آسانترین راه حل این است که
  - هردو مثبت باشد
  - به یاد داشته باشید که حاصل تکمیل شده اجرا می شود
  - علامت حاصلضرب را محاسبه می کند عملوندها را به اعداد مثبت تبدیل می کند ، بیت علامت را دور می ریزد ،برای31 مرتبه اجرا می شود ، سپس نتیجه ثابت می شود.

الگوریتم های سریعتر یا عمل ضرب

- الگوریتم بوت
  - ضرب اعداد علامت دار مانند قبل داز سخت افزار یکسان استفاده می کند و چرخه ها را ذخیره می کند
  - بیتهای مضرب را در یک لحظه می تواند دستکاری کند
- استفاده کردن از یک آرایه آدرس
- مشاهدات این گونه نشان می دهد که :چه اضافه شود یا نشود مضروب را یک واحد شیفت می دهیم_این تصمیم می تواند کامل به صورت موازی انجام شود.

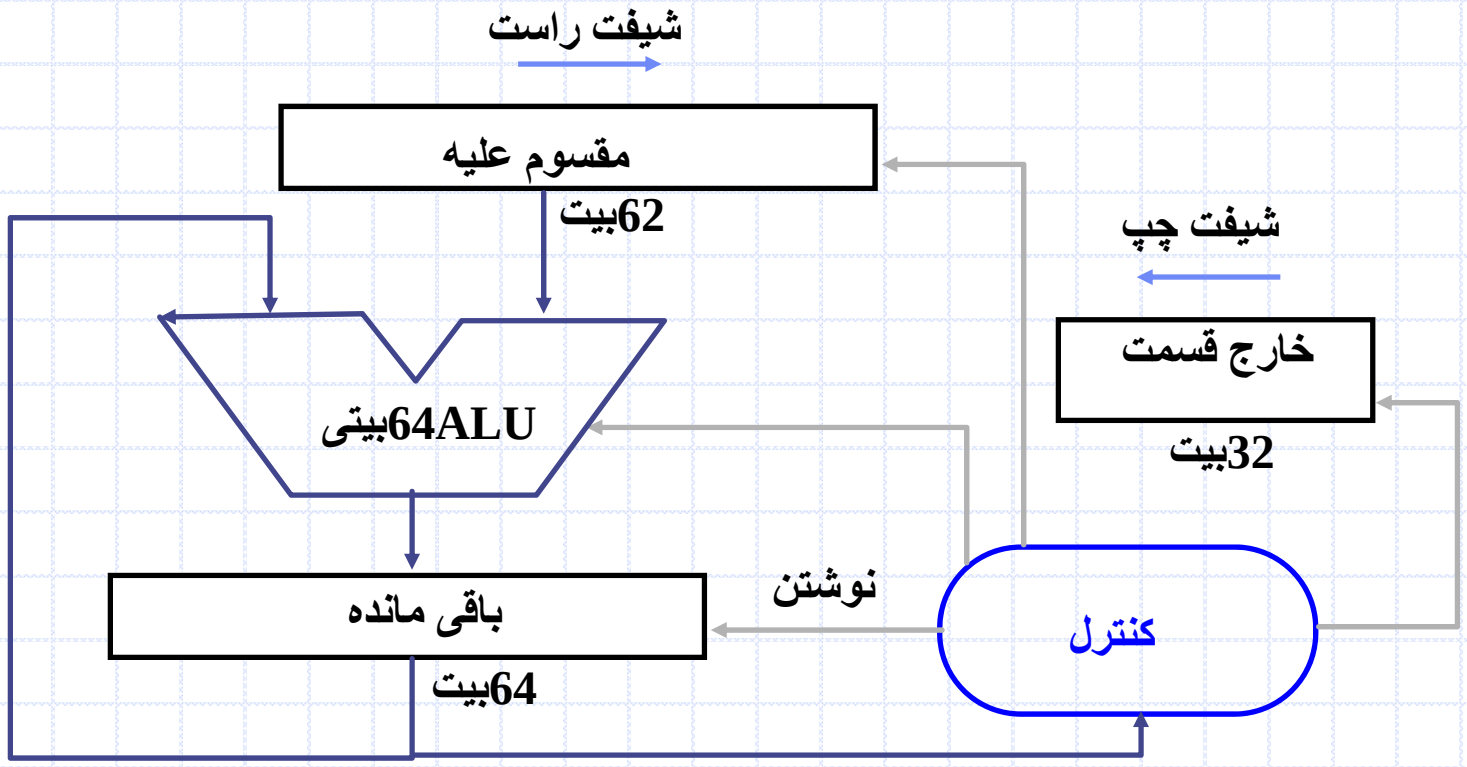
تقسيم :مداد و كاغذ

```
                1001   Quotient
Divisor 1000 | 1001010  Dividend
             -1000
               10
               101
               1010
              -1000
               10      Remainder
```

Dividend = Quotient * Divisor + Remainder

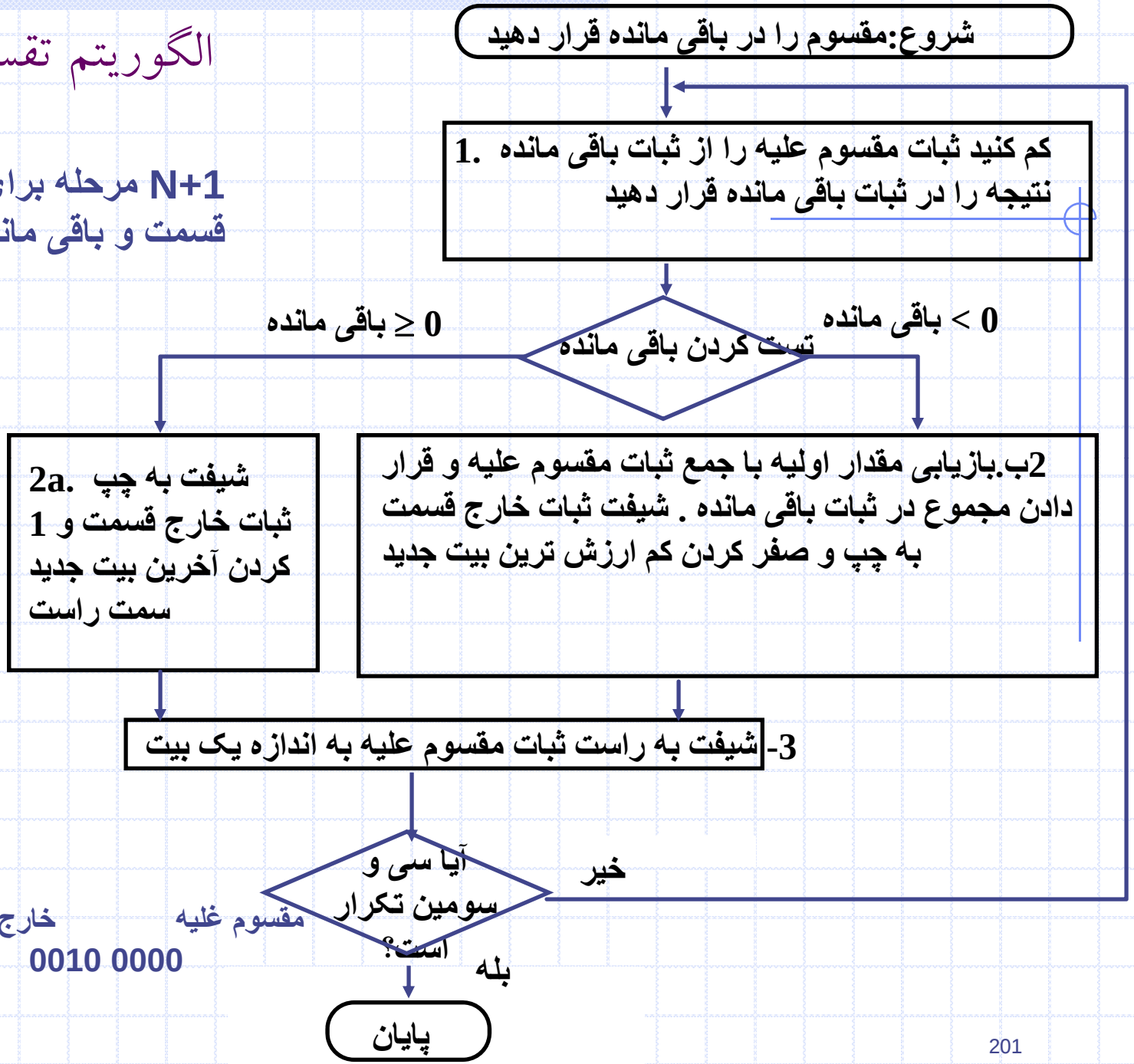
سخت افزار تقسیم ورژن1

- 64 بیت ثبات مقسوم علیه ،واحد محاسبه و منطق 64 بیتی ، ثبات باقی مانده ، ثبات خارج قسمت 32 بیتی



الگوریتم تقسیم ورژن 1

N+1 مرحله برای nبیت خرج قسمت و باقی مانده



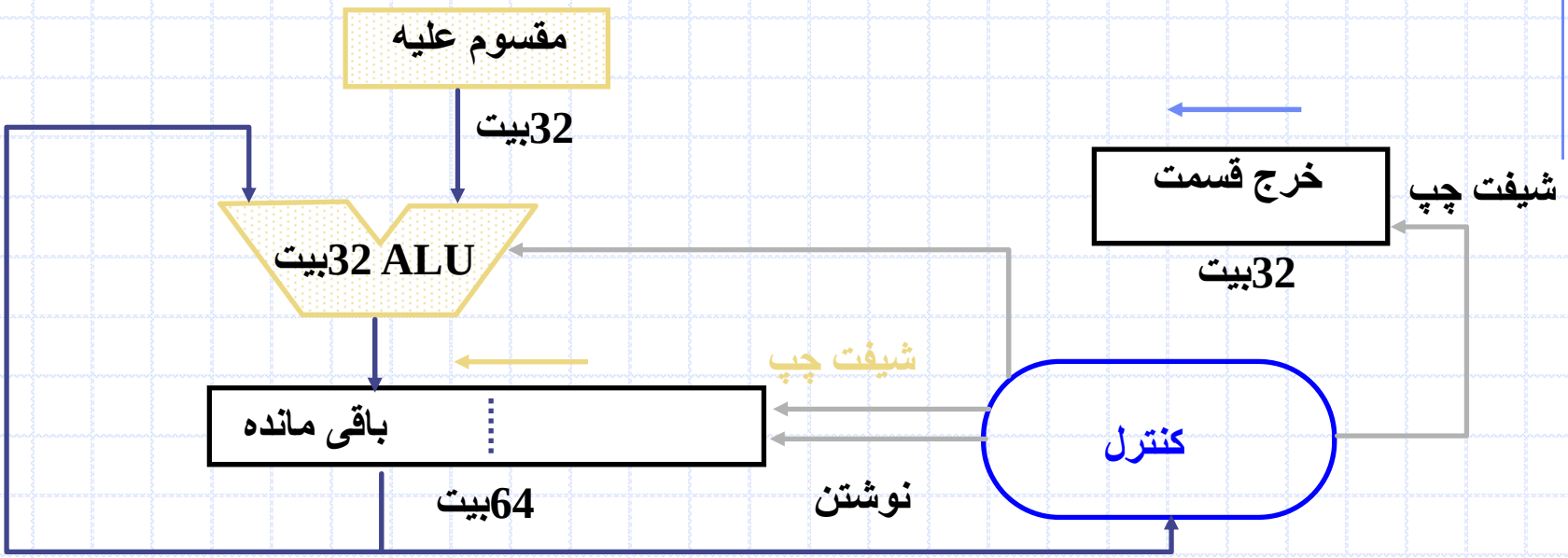
مقسوم غلیه	خارج قسمت	باقی مانده
0010 0000	0000	0000 0111

مشاهدات تقسیم ورژن 1

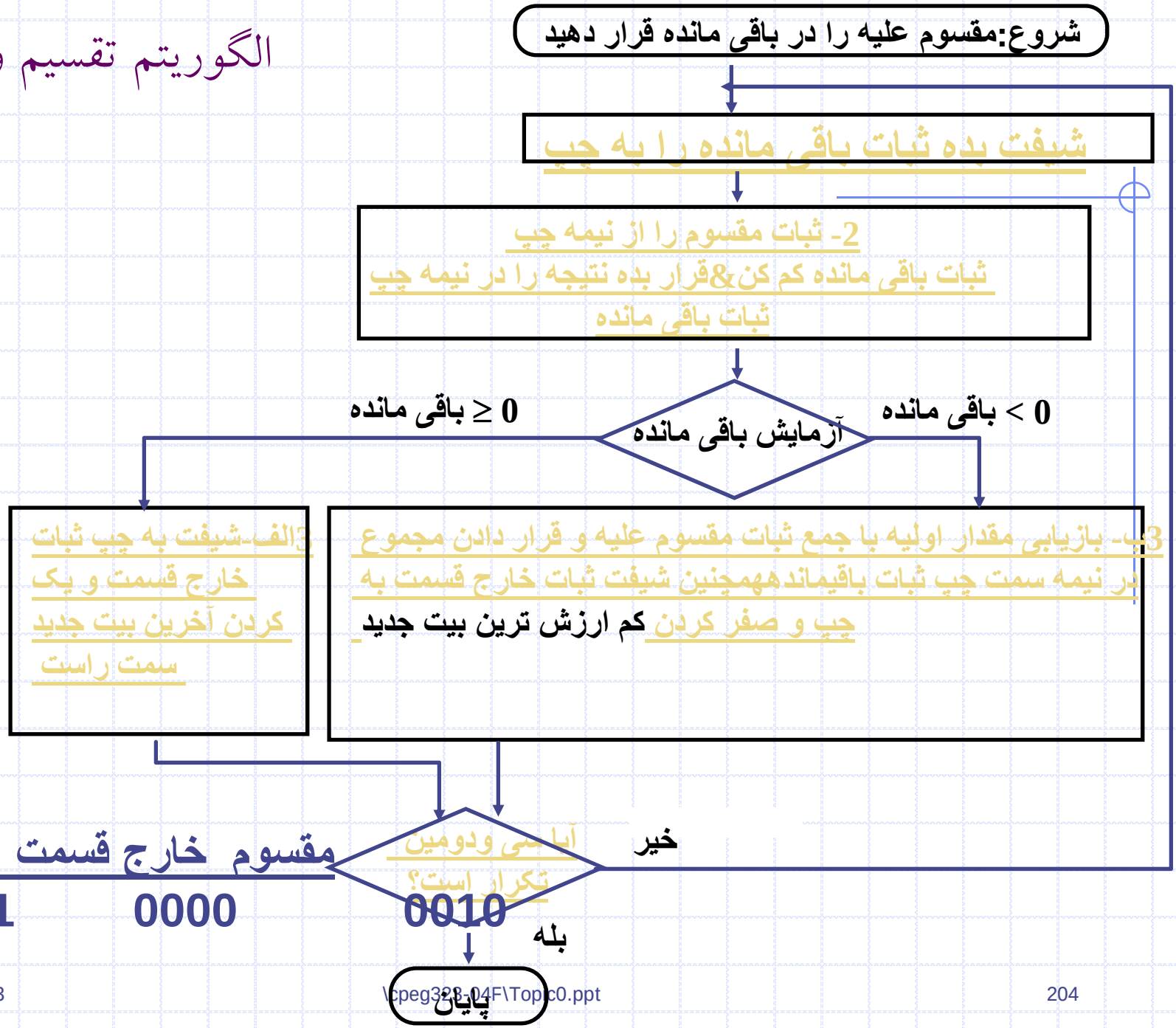
- نصف بیت ها در مقسوم علیه همیشه صفر است
- =>نصف 64 بیت جمع کننده هدررفته است
- =>نصف مقسوم علیه هدر رفته است
- به جای اینکه مقسوم علیه را شیفت به راست دهیم ، باقی مانده را شیفت به چپ می دهیم ؟
- مرحله اول نمی تواند تولید یک خارج قسمت کند(به عبارت دیگر خیلی بزرگ است)
- =>به وسیله اجرای فرمان ها با شیفت اول و سپس تفریق می توان یک تکرار ذخیره کرد.

سخت افزار تقسیم ورژن 2

- 32 بیت ثبات مقسوم علیه ، 32بیت ALU، 64بیت ثبات باقی مانده ،32بیت ثبات باقی مانده



الگوریتم تقسیم ورژن 2



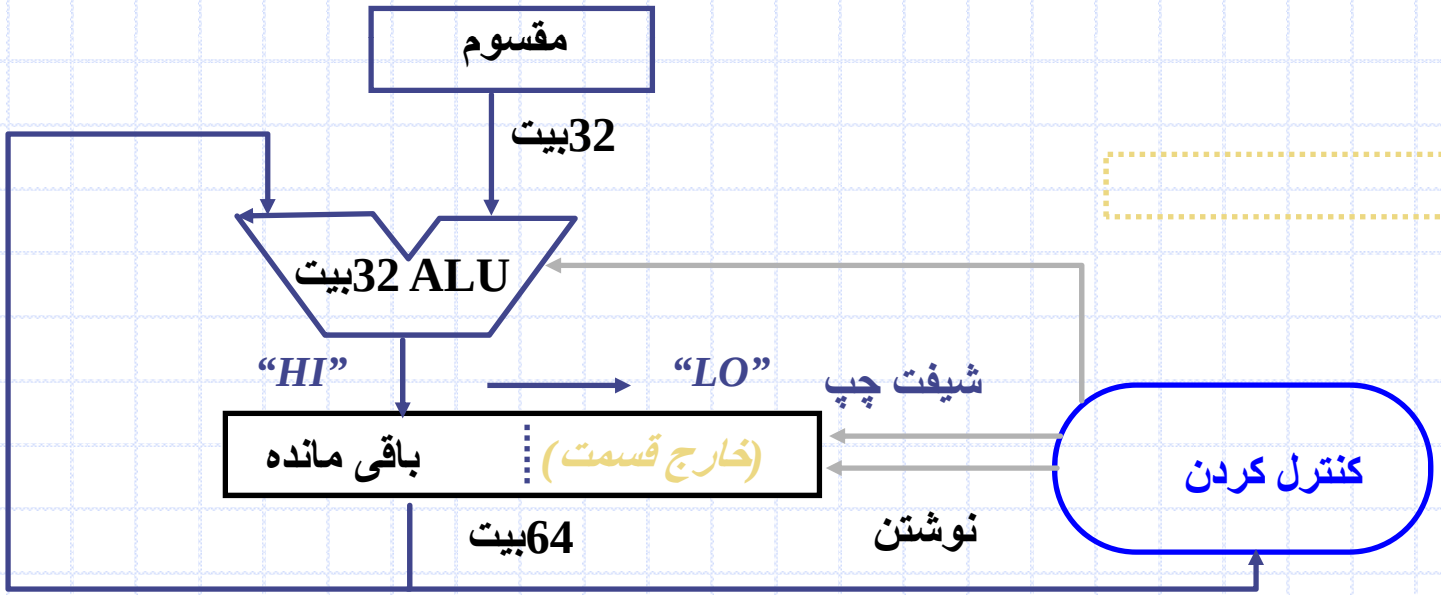
مقسوم	خارج قسمت	باقی مانده
0000 0111	0000	0010

مشاهدات تقسیم ورژن2

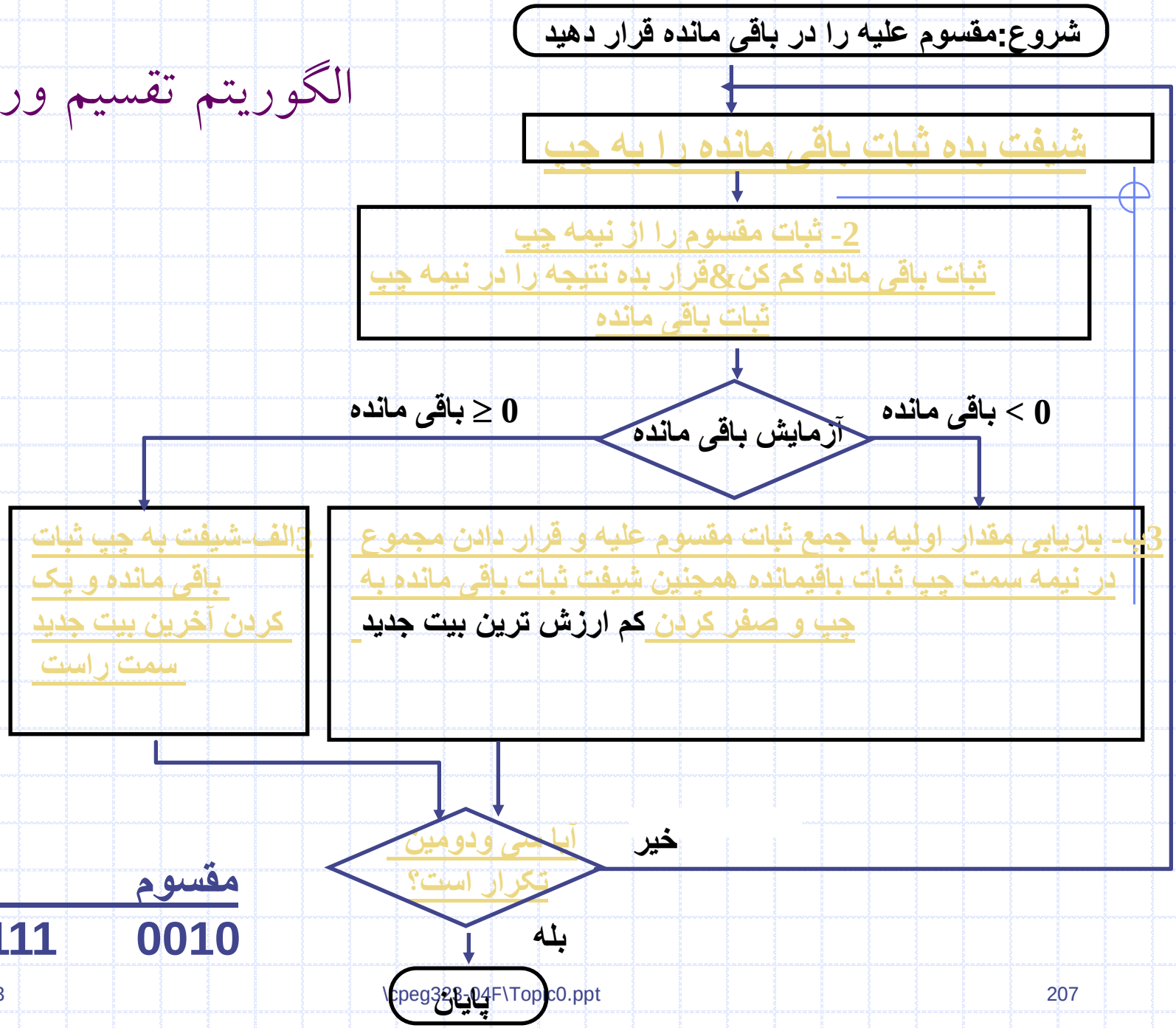
- حذف کردن ثبات خارج قسمت به وسیله بهم پیوستن با باقیمانده به عنوان شیفت چپ
- آغاز شیفت دادن باقی مانده به چپ مانند قبل
- پس از آن حلقه فقط محتوی 2 مرحله است چون شیفت دادن ثبات باقی مانده هر دو شیفت ها باقی مانده در نیمه چپ و خارج قسمت در نیمه راست.
- در نتیجه به هم پیوستن 2 ثبات به یکدیگر و فرمان های جدید عملگرها در این حلقه باقی مانده ای است که در یک لحظه بارها به چپ شیفت خواهد یافت
- بنابراین در مرحله تصحیح پایانی باید شیفت به عقب بدهیم که تنها باقی مانده در نیمه چپ ثبات باشد

سخت افزار تقسیم ورژن 2

- 32 بیت ثبات مقسوم ، 32بیت ALU، 64 بیت ثبات باقی مانده ، (صفر بیت خارج قسمت).



الگوریتم تقسیم ورژن 3



مقسوم	باقی مانده
0010	0000 0111

مشاهدات تقسیم ورژن3

- سخت افزار یکسان مانند ضرب:فقط به ALU نیاز داریم برای جمع یا تفریق ، 63 بیت شیفت به چپ یا راست
- ثبات های HIوlo به هم می پیوندند به عنوان ثبات 64 بیتی برای ضرب و تقسیم
- تقسیم علامت دار:وووووووووووووووووووووووووو
  - یادداشت:مقسوم علیه م باقی مانده باید هم علامت باشند
  - یادداشت:خارج قسمت منفی شده اگر علامت مقسوم و علامت باقی مانده مخالف باشند مثال :

-7÷2=-3 و باقی مانده=-1

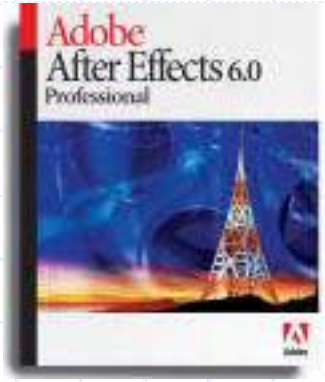
- ممکن است برای خارج قسمت خیلی بزرگ باشد: اگر تقسیم صحیح 64 بیتی به وسیله 1 ، خارج قسمت64 بیت است

فصل چهارم
سنجش کارایی

- سنجش: چه، چرا، چگونه.
- معادله کارایی
- قانون امدال
- چگونه انرژی کارایی را محدود می کند.

اغلب چه کسی CPUها را ارتقا می دهد

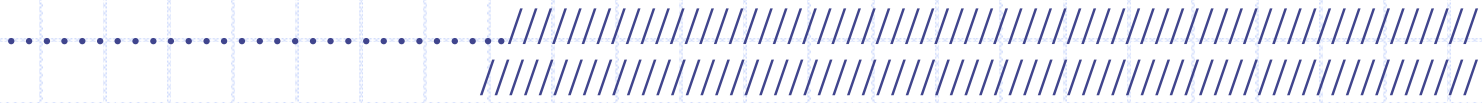
یک حرفه ای که چرخه های CPU را به پول تبدیل می کند، و محدود به چرخه است



Artist tool: animation, video special effects.

برای خرید یک ماشین جدید چگونه تصمیم می گیرید؟

اندازه گیری "زمان اجرای" After Effects در "ظرفیت کاری" تولید خروجی تصویر



Night flight Movie Goes Here

"Night flight"
نقشه شهر و ابرها
در آسمان بوسیله
تصاویر فرموله
محاسبه می شوند

CPUشدید
I/O ناچیز

کارایی و هزینه
از نظر مشتری/CPU

- از لحاظ خرید
  - معین کردن یک مجموعه از ماشینها که دارای
    - بهترین کارایی
    - کمترین هزینه
    - بهترین کارایی / هزینه
- از لحاظ طراحی
  - مواجه با انتخاب طراحی است که دارای:
    - بهترین بهبود کارایی
    - کمترین هزینه
    - بهترین کارایی / هزینه
- نیاز هر دو
  - پایه ای برای مقایسه
  - معیاری برای ارزیابی
- هدف ما فهماندن مفهوم هزینه و کارایی از انتخاب های معماری است.

دو تصور از "کارایی"

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concodre	3 hours	1350 mph	132	178,200

کدامیک کارایی بالاتری دارد؟

- **زمان انجام کار(زمان اجرا)**
  - زمان اجرا ، زمان پاسخ ، زمان برگشت
- **کارها در روز، ساعت ، هفته ، ثانیه ، نانو ثانیه ، ... (کارایی)**
  - کارایی ، توان عملیاتی ، پهنای باند

اغلب زمان پاسخ و توان عملیاتی در عکس هم قرار دارند، چرا؟

تعاریف کارایی

- کارایی میزان چیزهایی در ثانیه است
  - بزرگتر بهتر است
- اگر به صورت عمده، زمان پاسخ برای ما مهم باشد:
  - $$\text{performance}(x) = \frac{1}{\text{execution\_time}(x)}$$
- "X" n بار سریعتر از "Y" است به معنی اینست که :

$$n = \frac{\text{performance}(X)}{\text{execution\_time}(X)} = \frac{\text{execution\_time}(Y)}{\text{performance}(Y)}$$

- چه زمانی توان عملیاتی مهمتر از زمان اجراست؟
- چه زمانی زمان اجرا از توان عملیاتی مهمتر است؟

مثال هایی از کارایی

- زمانهای Concorde در مقابل Boeing 747 ؟
- Concord is 1350 mph / 610 mph = 2.2 بار سریعتر
  = 6.5 hours / 3 hours

- توان عملیاتی Concorde در مقابل Boeing 747 ؟
- Concord is 178,200 pmph / 286,700 pmph = 0.62 “times faster”
- Boeing is 286,700 pmph / 178,200 pmph = 1.6 “times faster”

- در مورد توان عملیاتی، Boeing 1.6 بار (”%60“) سریعتر است.
- در مورد زمان پرواز، Concorde 2.2 بار(”%120“) سریعتر است.

- زمانیکه بحث کارایی پردازنده مطرح است،عمده تمرکز ما بر روی زمان اجرا برای ”یک کار“ است. چرا؟

درک کارایی

- موارد زیر احتمالا، چگونه بر زمان پاسخ و توان عملیاتی اثر می گذارد؟
  - افزایش سرعت ساعت در یک پردازشگر معین.
  - افزایش تعداد کارها در یک سیستم (مثلا داشتن یک سرویس واحد کامپیوتری دارای چند کاربر).
  - افزایش تعداد پردازنده ها در یک سیستم که از چند پردازنده استفاده می کند (مانند یک شبکه از ماشینهای خود پرداز(ATM))
- اگر یک پنتیوم 3 یک برنامه را در 8 ثانیه اجرا کند و یک PowerPc برنامه ای مشابه را در 10 ثانیه اجرا کند، محصول پنتیوم چند بار سریعتر است؟

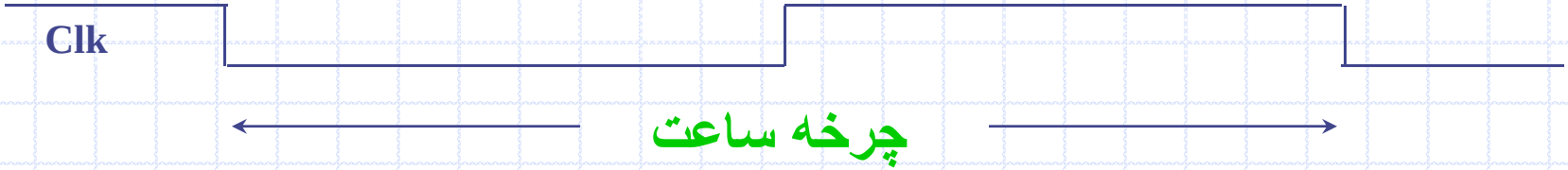
n = 10 / 8 = 1.25 times faster (or 25% faster)

- با این وجود چه چیزی باعث می شود که کسی PowerPC را برای خرید انتخاب کند ؟

تعریف زمان

- زمان به چند طریق قابل تعریف است، وابسته به اینکه چگونه مورد سنجش قرار دهیم:
  - زمان پاسخ: مجموع زمان صرف شده برای تکمیل یک کار، شامل زمان صرف شده برای اجرا در CPU دستیابی به دیسک و حافظه، انتظار برای I/O و دیگر پردازشها، و سربار سیستم عامل.
  - زمان اجرای CPU: مجموع زمانی که یک CPU برای یک کار صرف می کند(به غیر از زمان I/O یا اجرای برنامه های دیگر). این زمان به طور ساده معروف به زمان CPU است.
  - زمان کاربر CPU: مجموع زمانهایی که CPU در برنامه صرف می کند.
  - زمان اجرای سیستمی CPU : مجموع زمانی که سیستم عامل صرف اجرای کارها در برنامه می کند.
- برای مثال: یک برنامه می تواند دارای زمان سیستمی CPU معادل 22 ثانیه باشد، یک زمان کاربر CPU برابر 90 ثانیه ، یک زمان اجرایی CPU، معادل 112 ثانیه ، و یک زمان پاسخ برابر 162 ثانیه باشد.

ساعت کامپیوتر



- یک ساعت کامپیوتر در یک نرخ ثابت انجام می گیرد و تعیین می کند چه زمانی رخدادها در سخت افزار قرار گیرند.
  - زمان چرخه ساعت مقدار زمان برای سپری شدن یک چرخه ساعت است (مثلا 5 نانوثانیه).
  - نرخ ساعت، عکس زمان چرخه ساعت است.
  - مثلا، اگر یک کامپیوتر دارای زمان چرخه ساعت معادل 5 نانوثانیه باشد، نرخ ساعت برابر است با:

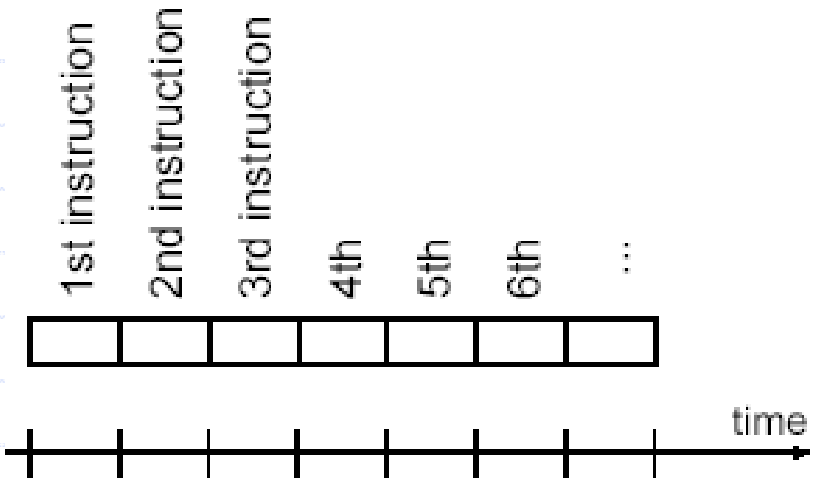
$$\frac{1}{5 \times 10^{-9}\ \text{sec}} = 200\ \text{MHz}$$

معیارهای پایه ارزیابی

- مقایسه معیارهای ماشینها
  - زمان اجرا
  - توان عملیاتی
  - زمان CPU
  - MIPS : مخفف : میلیونها دستورالعمل در ثانیه
  - MFLOPS : میلیونها عمل ممیز شناور در هر ثانیه
- مقایسه مجموعه برنامه های استفاده شده در ماشینها
  - مفهوم محاسباتی
  - مفهوم محاسباتی سنگین
  - Benchmarks

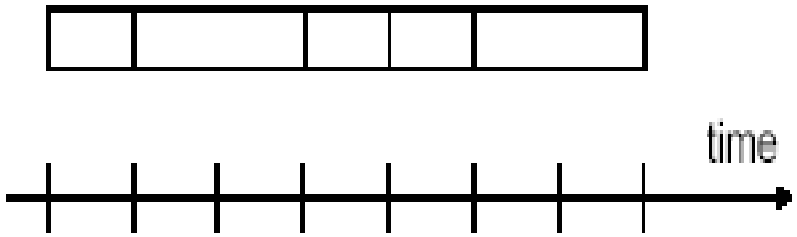
چه تعداد چرخه برای یک برنامه نیاز است؟

ممکن است تصور کرده باشید که تعداد چرخه ها = تعداد دستورالعملها باشد



این تصور اشتباه است، دستورالعملهای متفاوت ، در ماشینهای متفاوت زمانهای متفاوتی را صرف می کنند.

تعداد چرخه های متفاوت برای دستورالعمل های متفاوت



- تقسیم زمان خیلی بیشتری از جمع صرف می کند.

اعمال ممیز شناور، زمان بیشتری نسبت به نوع صحیح آن مصرف می کند.

دسترسی به حافظه زمان بسیار بیشتری از دسترسی به ثباتها صرف می کند.

اکنون که ما با مفهوم چرخه آشنا شدیم

- یک برنامه معین نیاز دارد به
  تعدادی از دستورالعملها
  تعدادی از چرخه های ساعت
  تعدادی از ثانیه ها

- ما یک لغت نامه برای شرح این کمیت ها داریم:
  زمان چرخه ساعت (ثانیه در هر چرخه)
  نرخ ساعت (سیکل در ثانیه)
  CPI(تعداد چرخه ها در هر دستورالعمل)

یک برنامه ممیز شناور ممکن است دارای CPI بالاتری باشد

محاسبه زمان CPU

- زمان اجرای یک برنامه معین می تواند مانند زیر محاسبه شود

زمان چرخه ساعت x چرخه های ساعت CPU = زمان CPU

از آنجا که زمان چرخه ساعت و نرخ ساعت عکس یکدیگرند

نرخ ساعت/چرخه های ساعت CPU=زمان CPU

- تعداد چرخه های ساعت CPU می تواند مانند زیر محاسبه گردد

CPUچرخه های ساعت) = دستورالعملها/برنامه) x (چرخه های ساعت/دستورالعمل(

= تعداد دستورالعملها x CPI

بنابراین

CPUزمان = تعداد دستورالعملها x CPI x زمان چرخه ساعت

CPUزمان = تعداد دستورالعملها / CPI x نرخ ساعت

- واحدهای این، اینگونه هستند

$$\text{ثانیه} = \frac{\text{دستورالعملها}}{\text{برنامه}} \times \frac{\text{چرخه های ساعت}}{\text{دستورالعمل}} \times \frac{\text{ثانیه}}{\text{چرخه ثانیه}}$$

مثالی از محاسبه زمان CPU

- اگر یک کامپیوتر دارای نرخ ساعت 50MHZ باشد، چقدر طول می کشد تا یک برنامه دارای 1000 دستورالعمل اجرا شود؟ CPI برای این برنامه 3.5 است.
- استفاده از معادله

زمانCPU =تعداد دستورالعمل / CPI x نرخ ساعت

پس

زمانCPU(10×50) / 3.5×1000 = 10^6

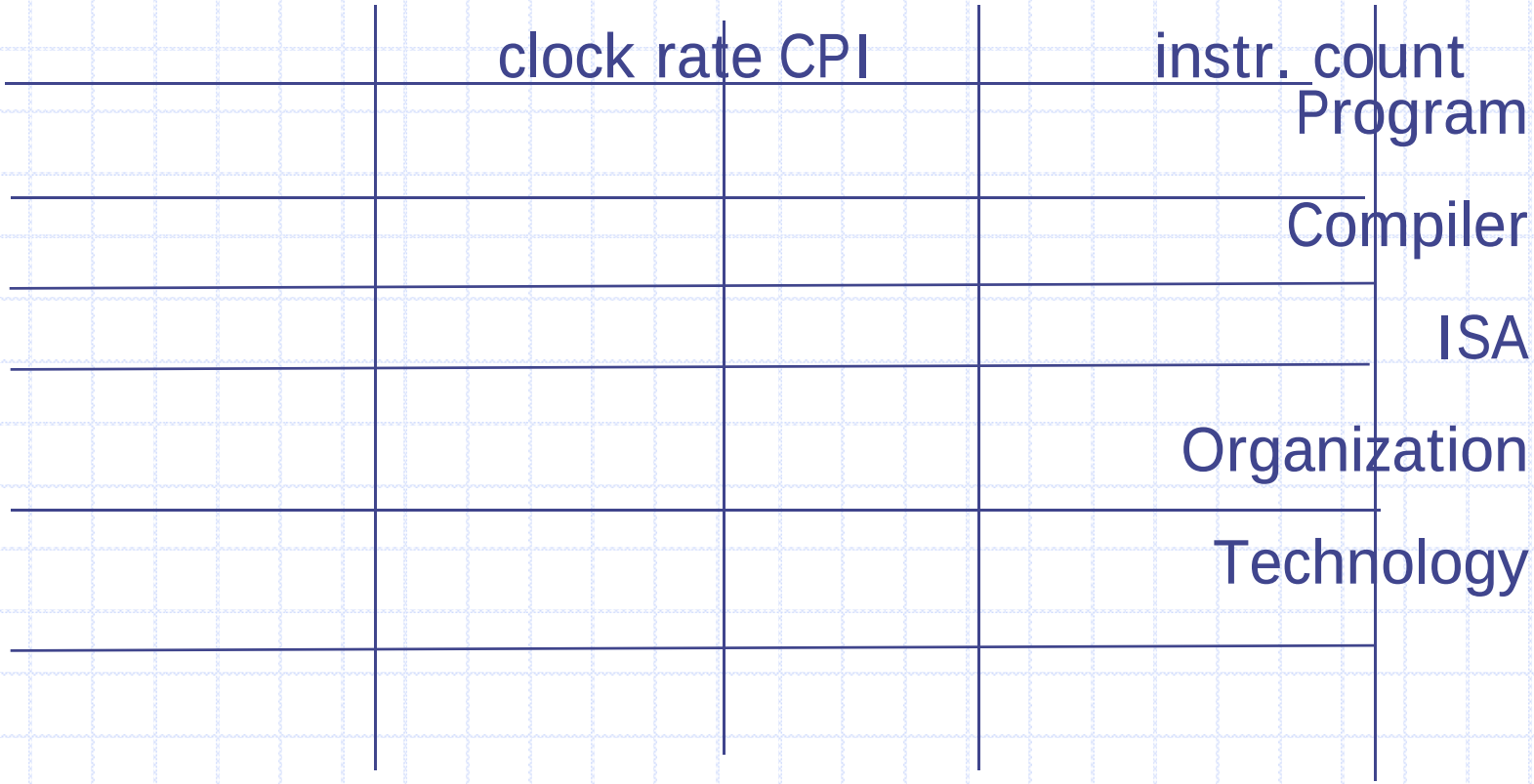
- اگر نرخ ساعت CPU ی یک کامپیوتر از 200MHZ به 250 MHZ افزایش یابد و بقیه فاکتورها یکسان باقی بمانند، افزایش سرعت کامپیوتر چقدر است؟

$$\frac{\text{زمان CPUقدیمی}}{\text{زمان CPUجدید}} = \frac{\text{نرخ جدید ساعت}}{\text{نرخ قدیمی ساعت}} = \frac{250\ \text{MHz}}{200\ \text{MHZ}} = 1.25$$

- تصور ساده انگارانه ما چه بود؟

فاکتورهای موثر در کارایی CPU

- چه فاکتورهایی از موارد زیر تاثیر می گیرند؟



$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

فاکتورهای موثر در کارایی CPU

- چه فاکتورهایی از موارد زیر تاثیر می گیرند؟

	clock rate	CPI	instr. count
Program	*	*	*
Compiler	*	*	*
ISA	*	*	*
Organization	*	*	*
Technology	*		

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

محاسبه CPI

- CPI،میانگین تعداد چرخه ها در هر دستورالعمل است
- اگر برای هر نوع دستورالعمل ، لازم است ما فرکانس ها و تعداد چرخه ها را برای اجرای آن بدانیم ، ما می توانیم CPI را به صورت زیر محاسبه کنیم: i

$$CPI = \Sigma CPI_i \times F_i$$

- برای مثال:

Op	F_i	CPI_i	$CPI_i \times F_i$	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%
Total	100%		2.2	100%

مسائل با میانگین حسابی

- تقاضاهایی که احتمال یکسانی برای اجرا ندارند. 2 تا ماشین که زمان بندی شدن برای2 آزمون کارایی:

	Machine A	Machine B
Program1	2 seconds(%20)	6 seconds
Program2	12 seconds(%80)	10 seconds

میانگین زمان اجرا=(12+2)/2 =7ثانیه
میانگین زمان اجرا=(10+6)/2=8 ثانیه

میانگین زمان اجرای وزنیA=12*0.8 +2*0.2=10 ثانیه
میانگین زمان اجرای وزنیB=10*0.8 + 6*0.2=9.2ثانیه

خلاصه ای از کارایی

- 2 سنجش اصلی کارایی
- زمان اجرا:زمان انجام کار
- توان عملیاتی:تعداد کارهایی هستند که در یک واحد زمانی کامل می شوند
- کارآیی و زمان اجرا متقابل اند افزایش کارایی کاهش زمان اجرا را در پی دارد
- زمان اجرایی که به یک برنامه اختصاص داده می شود می تواند به صورت زیر محاسبه شود

زمان CPU = تعداد دستورالعمل ها * CPI * زمان هر چرخه

زمان CPU= تعداد دستورالعمل ها * CPI/آهنگ پالس ساعت

این عوامل متآ ثر است از تکنولو ژی کامپایلر ، معماری مجموعه ذستور العمل ها ، سازمان دهی ماشین ، و تکنولو ژی زیر بنایی .زمانی که کارایی افزایش می یابد دقت کنید که چه رخدادهایی به طور متناوب اتفاق می افتد =><نمونه متداول سریعی ساخته می شود.

به خاطر داشته باشید: بخشهای اشتراکی را سریعتر نمائید!

آزمون های کارایی کامپیوتر

- آزمون کارایی ، یک برنامه یا مجموعه ای از برنامه هایی است که برای دستیابی کامپیوتر به کارایی استفاده می شود
- آزمون های کارایی به ما اجازه می دهند سنجش کارایی بر پایه زمان های اجرایی را می دهد
- آزمون های کارایی باید نمایش دهنده نوع درخواست های در حال اجرا در یک کامپیوتر باشند
- آزمون های کارایی نباید به عوامل دیگری در کامپیوتر وابسته باشند ، مثل موس ، کیبورد ، و...
- آزمون های کارایی می توانند در گونه های پیچیده و سودمندشان بسیار گوناگون باشند.

انواع آزمون های کارایی

مزایا		معایب
• نمایشگر	هدف اصلی بار کاری	• خیلی مشخص • غیر قابل حمل • برای اجرا یا سنجش سخت است • برای همانندسازی سخت است
• قابل حمل • کاملا کاربردی • پیشرفت های مفید در واقعیت	درخواست کامل آزمون های کارایی (مثل آزمون های کارایی spec)	• نشان دهنده کم
• آسان برای اجرا در اوایل طراحی چرخه	Small "Kernel" Benchmarks	• سیستم حافظه قابل اندازه گیری نیست
• قابلیت به اوج رسانیدن همانندی و پتانسیل گلوگاه ها	آزمون های کارایی میکرو	• به اوج رسیدن"می تواند یک راه طولانی از درخواست کارایی باشد

SPEC

(System Performance Evaluation Cooperative)

موسسه ارزیابی کارایی سیستم

- آزمون های کارایی SPECبه شدت برای گزارش کارایی و کارایی PCبه کار می رود
- اولین دوره SPEC CPU89
  - شامل 10 برنامه روی اعداد single
- دومین دوره SPEC CPU92
  - Spec CINT92(6 تا برنامه صحیح) وSpec CFP92(14 تا برنامه ممیز شناور)
  - فلگ های کامپایلر می توانند قرار بگیرند به طور متفاوت برای برنامه های متفاوت
- سومین دوره SPEC CPU95
  - قرار گیری جدید برنامه ها:SPEC CINT 95(8 تا برنامه صحیح) و SPEC CFP 95(10 تا برنامه ممیز شناور)
  - تنها فلگ کامپایلری که قرار می گیرد برای همه برنامه ها .

SPEC

- چهارمین دوره SPEC CPU2000
- قرار گیری جدید برنامه ها :SPEC CINT2000(12 تا برنامه صحیح) و SPEC CFP2000 (14 تا برنامه ممیز شناور)
- تنها فلگ کامپایلری که قرار می گیرد برای همه برنامه ها .
- گزارشات مستند نسبت ارزیابی برای SPEC هستند
- زمان CPU ماشین مورد ارزیابی/زمان CPU ماشین مرجع

دیگر آزمون های کارایی SPEC

- JVM98:
  - اندازه گیری کارایی ماشین های مجازی
- SFS97:
- اندازه گیری قوانین فایل سرویس دهند شبکه (NFS)
- Web99:
- اندازه گیری کارایی در خواست های شبکه گسترده جهانی
- HPC96:
- اندازه گیری کارایی گسترده در خواست های صنعتی
- APC,MEDIA,OPC
- اندازه گیری کارایی درخواست های گرافیکی
- برای اطلاعات بیشتر درباره آزمون های کارایی SPEC به http://www.spec.org مراجعه کنید

مثال هایی از آزمون های کارایی SPEC95

- نسبت های SPECنشان داده می شوند درپردازش گرهای Pentiumو Pentium pro(pentium +)

Clock Rate	Pentium SPECint	Pentium+ SPECint	Pentium SPEFfp	Pentium+ SPECfp
100 MHz	3.2	N/A	2.6	N/A
150 MHZ	4.4	6.0	3.0	5.1
200 MHZ	5.5	8.0	3.8	6.8

ازاین اطلاعات چه چیزی را می توان فرا گرفت؟

عوامل ضعیف ارزیابی کارایی

- اندازه گیری کارایی کامپیوتر شامل MIPSوMFLOPS می شود
- MIPS: میلیون ها دستورالعمل در هر ثانیه

_MIPS=(زمان اجرا*10^6)/تعداد دستورالعمل ها

_برای مثال یک برنامه که 3 میلیون دستورالعمل را در 2ثانیه محاسبه می کند درجه MIPSآن 1.5است

_مزایا:آسان است برای فهمیدن و اندازه گیری

_معایب:ممکن است کارایی واقعی را منتقل نکند در حالی که دستورالعمل های ساده تر را بهتر اجرا می کند.

- MFLOPS:میلیون ها عملگر ممیز شناور در هر ثانیه

_MFLOPS=(زمان اجرا*10^6)/عملگرهای ممیز شناور

_برای مثال برنامه S که 4 میلیون دستورالعمل را در 5ثانیه محاسبه می کند درجه MFLOPSآن 0.8 است.

_مزایا:آسان است برای فهمیدن و اندازه گیری

_معایب :مانند MIPSتنها اندازه گیری های ممیز شناور

بالا بردن میزان تسریع به صورت زیر تعریف می شود

میزان تسریع= زمان اجرای قدیم / زمان اجرای جدید = کارایی جدید / کارایی قدیم

- این عامل انکسار و انحراف را برای میزان تسریع چنین در نظر می گیریم

$$\text{زمان اجرای قدیم} = \text{زمان اجرای جدید} \times \frac{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{میزان تسریع افزوده شده}}}{1}$$

$$\text{میزان تسریع} = \frac{\text{زمان اجرای قدیم}}{\text{زمان اجرای جدید}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{میزان تسریع افزوده شده}}}$$

مثالی از قانون آمدال

- دستورالعمل های ممیز شناور برای 2 برابر شدن سرعت اجرا بهبود داده شده اند ، اما تنها 10%زمان برای این دستورالعمل های اصلی استفاده می شود. ماشین جدید چقدر سریعتر است؟

$$\text{میزان تسریع} = \frac{\text{زمان اجرای قدیم}}{\text{زمان اجرای جدید}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$\text{میزان تسریع} = \frac{1}{(1 - 0.1) + 0.1/2} = 1.053$$

ماشین جدید 1.053سرعت دارد ,یا 5.3%سریعتر است

اگر سرعت دستورالعمل های ممیز شناور 100 برابر سریعتر شود ماشین جدید چقدر سریعتر می شود؟

$$\text{میزان تسریع} = \frac{1}{(1 - 0.1) + 0.1/100} = 1.109$$

تخمین بهبود کارایی

- فرض کنید یک پردازشگر به طور جاری به 10ثانیه برای اجرا ی یک برنامه نیاز دارد و کارایی یک پردازشگر هر سال 50%بهبود می یابد.
- به وسیله چه عاملی کارایی پردازشگر در 5 سال بهبود پیدا می کند؟

(1+0.5)^5=7.59

بعد از 5 سال یک پردازشگر همان برنامه را در چه زمانی اجرا می کند؟ زمان اجرای جدید=10/7.59=1.32 ثانیه

چه فرضیاتی در مسئله بالا پدید می آیند؟

مثال کارایی

- کامپیوتر های M1وM2اجراهای مجموعه دستورالعمل های یکسانی دارند
M1آهنگ پالس ساعت 50MHzدارد و M2آهنگ پالس ساعت 75MHzدارد.M1دارای CPI 2.8است و M2دارایCPI 3.2 برای برنامه دارد.
چقدر زمان M2سریعتر است نسبت به M1در این برنامه؟

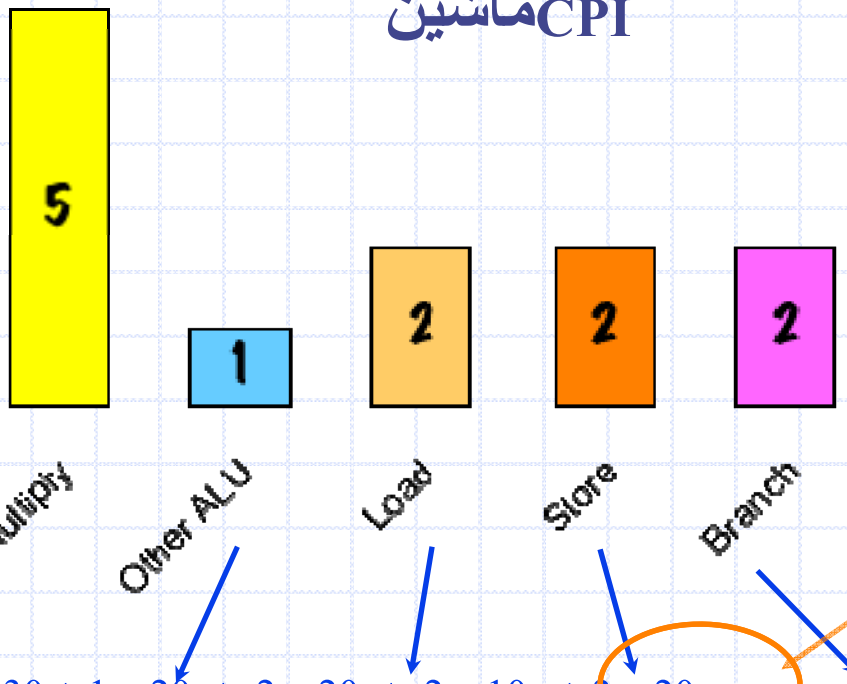
زمان اجرایM1 = تعداد دستورالعملها M1در x CPI_{M1} / M1آهنگ پالس ساعت = 2.8/50 = 1.31

زمان M2 تعداد دستورالعمل ها x CPIM2 / M2آهنگ پالس ساعت 3.2/75

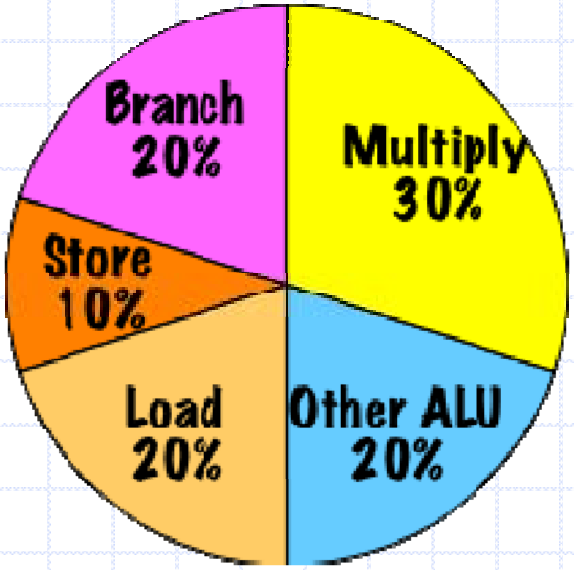
آهنگ پالس ساعت M1چقدر باید باشد تا زمان اجرای یکسانی برای آنها داشته باشد؟

CPIبه عنوان یک ابزار تحلیلگر برای راهنمای طراحی

CPIماشین

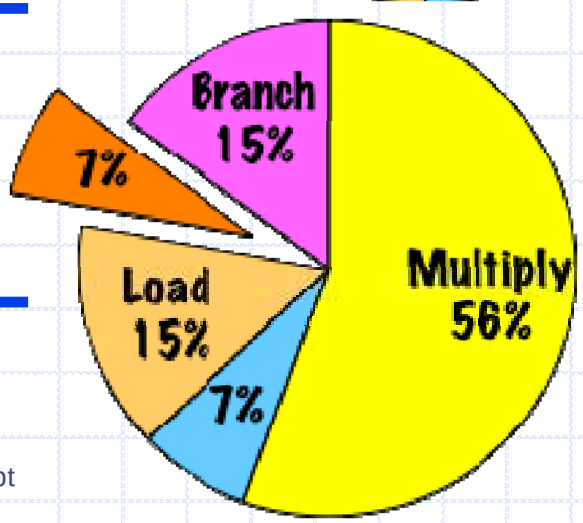


میکس دستورالعمل های برنامه



$$\frac{5 \times 30 + 1 \times 20 + 2 \times 20 + 2 \times 10 + 2 \times 20}{100}$$

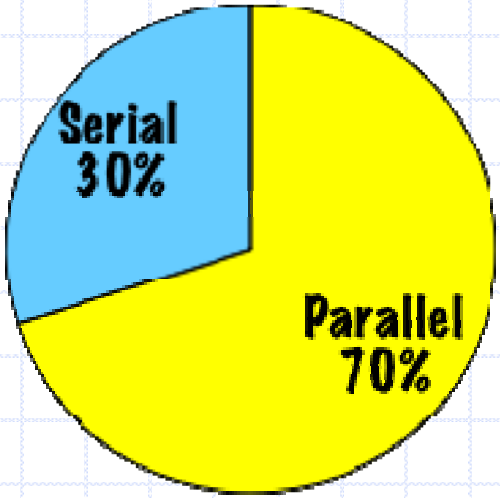
= 2.7 cycles/instruction



$$\frac{5 \times 56 + 1 \times 7 + 2 \times 15 + 2 \times 7 + 2 \times 15}{100}$$

= 3.61 cycles/instruction

دستورالعملPEER :قانون آمدال



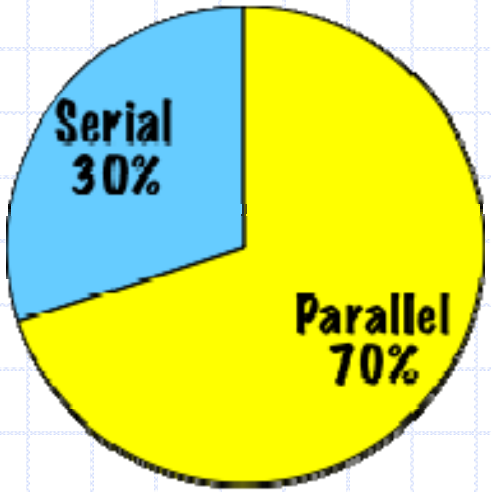
برنامه 30% از زمانش را به کدهایی اختصاص می دهد که نمی توانند دوباره برای اجرا به صورت موازی کد شوند

میزان تسریع محاسبه می شود برای

N = 2, 3, 4, 5, and ∞.

CPUs	2	3	4	5	∞
میزان تسریع					

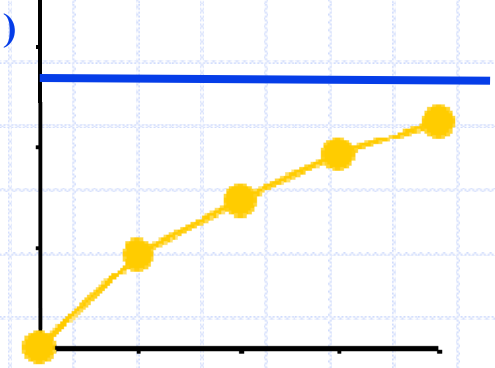
دستور العمل PEER: قانون آمدال



برنامه 30% از زمانش را به صورت سریال استفاده می کند

میزان تسریع را برای N = 2, 3, 4, 5, ∞. محاسبه می کند

$$S = \frac{1}{30\ \% + (70\% / N)}$$



CPUs	2	3	4	5	∞
میزان تسریع	1.54	1.85	2.1	2.3	3.3

یک ژول انرژی :یک جریان 1 آمپری است که از وسط یک مقاومت 1 اهمی عبور می کند و گرمایی معادل 1 ژول تولید می کند

و همچنین یک وات برای یک ثانیه

1 وات :عبور 1 آمپر جریان از یک مقاومت 1 اهمی

انرژی و کارایی

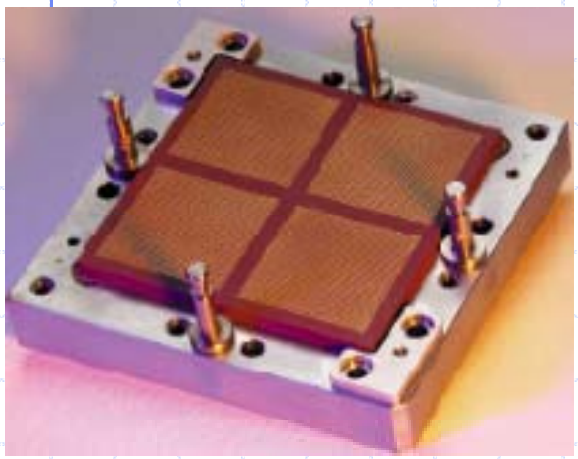
کالری 0.24 = یک **ژول**

یک کالری دمای یک گرم آب را یک درجه افزایش می دهد

یک مورد ناخواسته:کامپیوترها انر ژیالکتریکی را به گرما تبدیل می کنند. که به عنوان فراورده های جانبی محاسبه می شود .

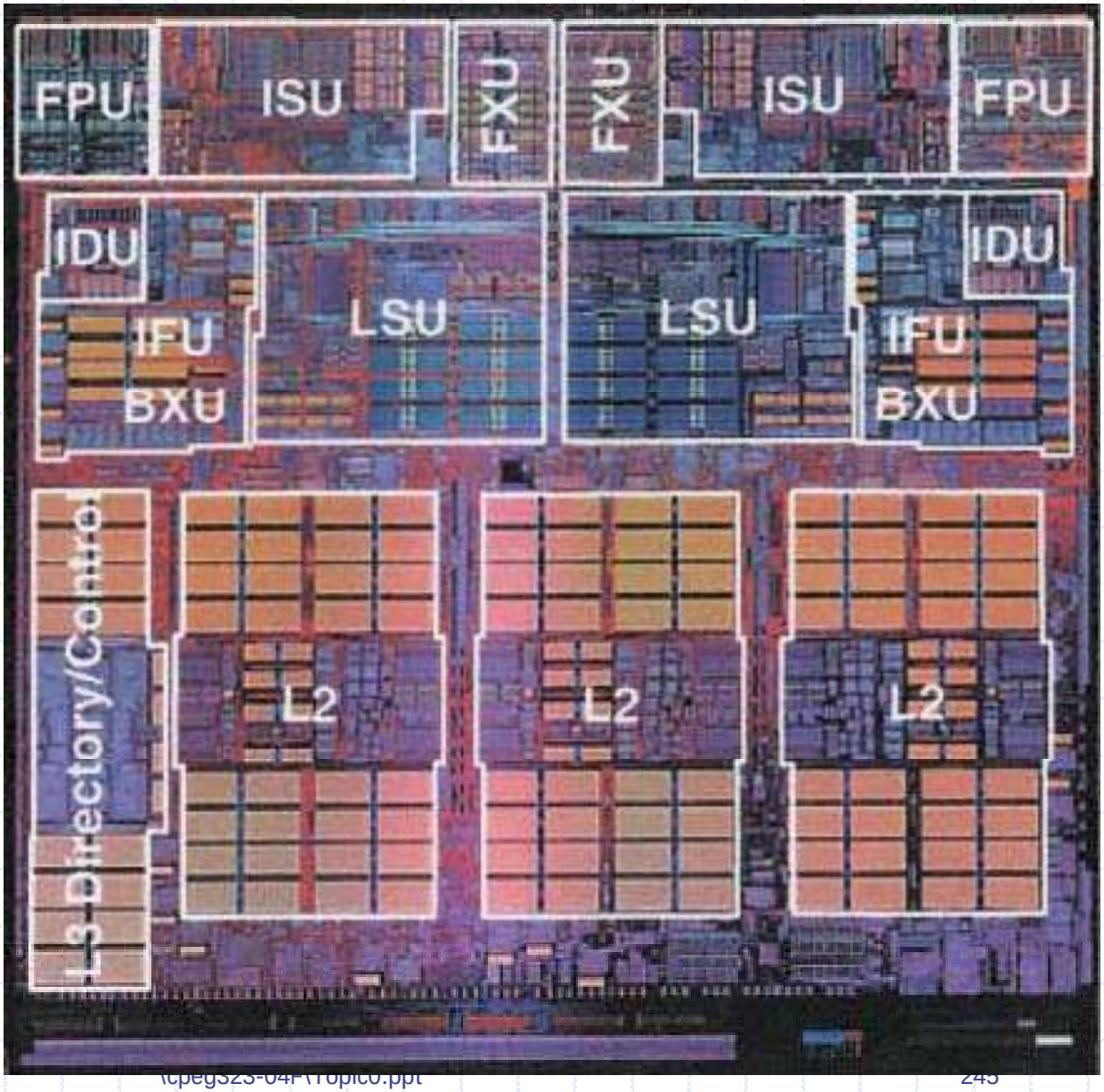
هوا یا آب می توانند مانع ذوب شدن چیپ در اثر انتقال حرارت شود

IBM Power 4: How does die heat up?

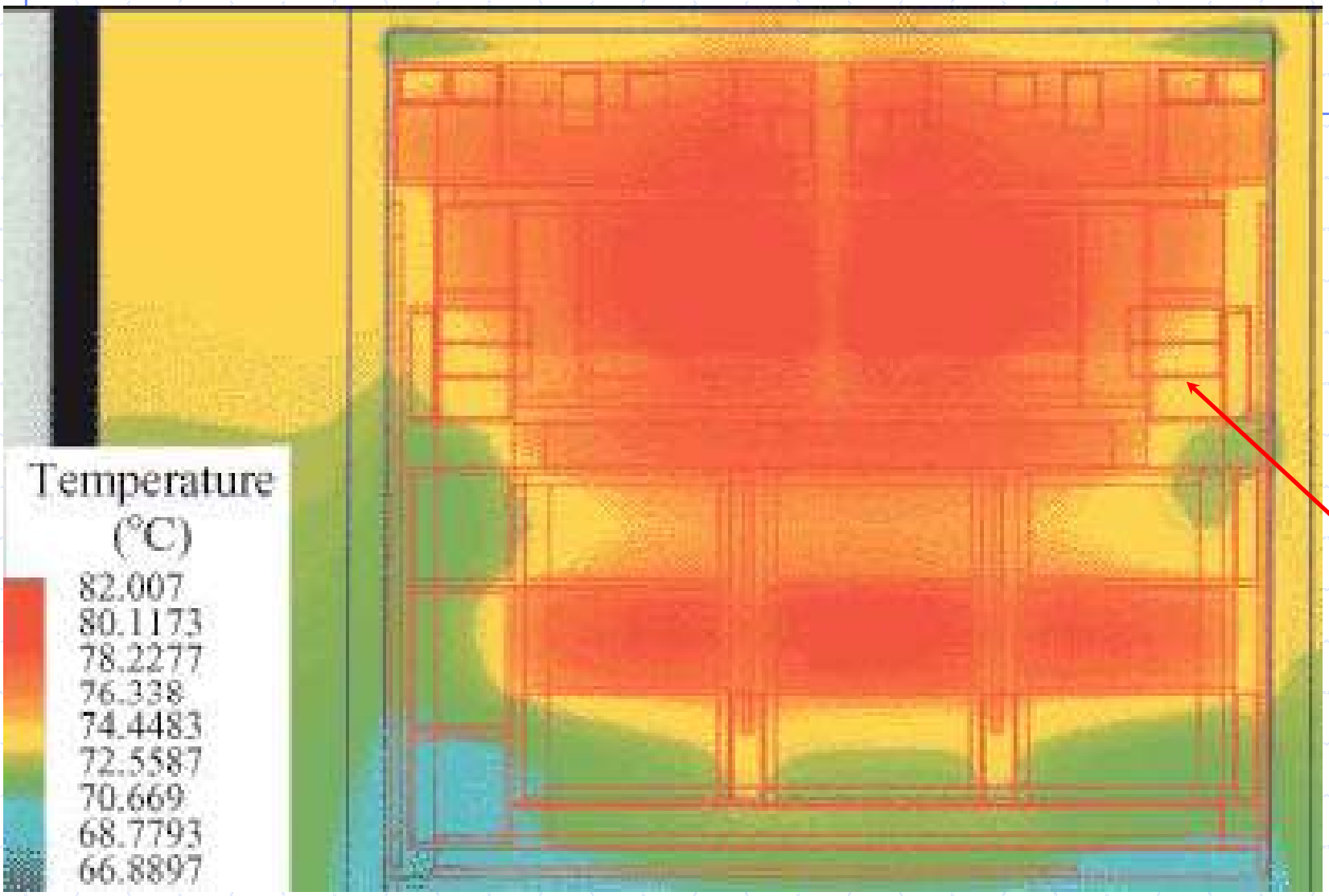


4 dies on a multi-chip module

2 CPUs per die



IBM Power 4: Dissipating 115 Watts



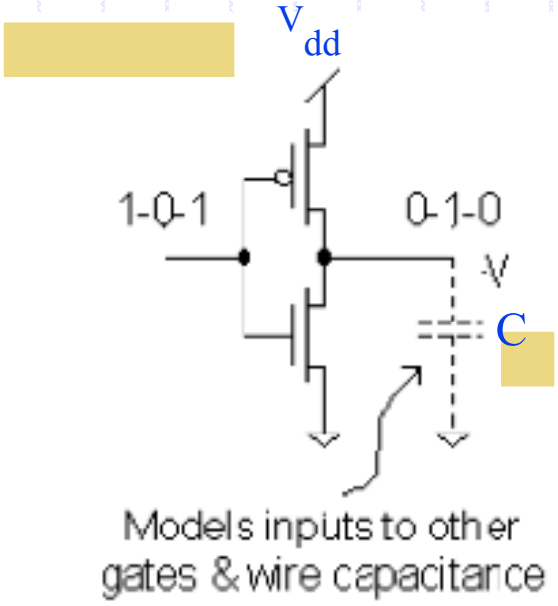
Hot spots

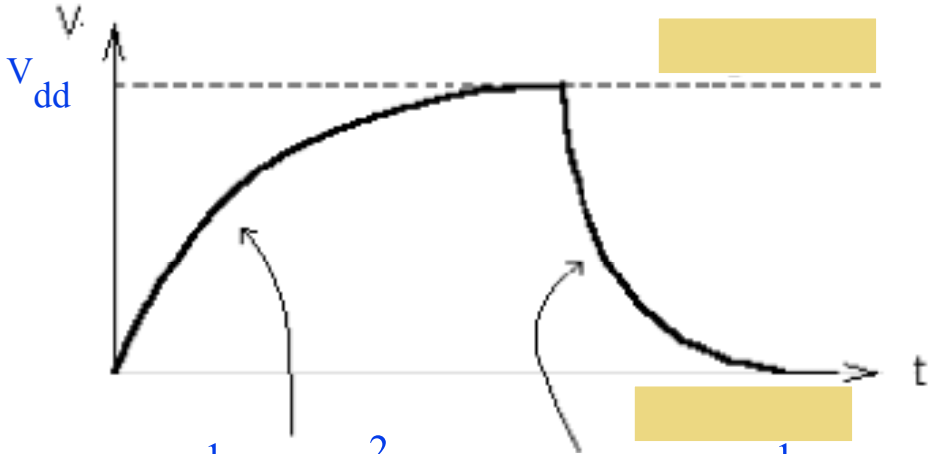
Fixed point units

Cache logic

اتصال انرژی :فیزیک های بنیادی

انتقال منطقی اتلاف انرژی





$$E_{0->} = \frac{1}{2} C V_{dd}^2 \qquad E_{1->} = \frac{1}{2} C V_{dd}^2$$

نتیجه قطعی:مستقل از تکنولو ژی

State-of-the-art CPUs (90 nm):

Switching energy is 70% of total energy.

Remainder: at 90nm, “switches” are “dimmers”!

How can we limit switching energy?

خلاصه ای از ارزیابی کارایی

- آرمون های کارایی خوب از قبیل آزمون های کارایی SPEC می تواند روش خاصی را برای ارزیابی و مقایسه کارایی کامپیوتر تولید کند
- MIPS و MFLOPS برای استفاده آسان هستند اما برای نشان دادن کارایی نا صحیح هستند.
- قانون آمدال یک روش موثر را برای توضیح میزان تسریع وابسته به عامل انحراف ایجاد می کند.

به خاطر داشته باشید: بخشهای اشتراکی را سریعتر نمائید!

نتیجه گیری

- مشتری ها:قیمت را می سنجند
  معمارها : طرح را می سنجند
- ابزارآلات:CPIو معادله کارایی
- قانون آمدال:محاسبه می کند
- انرژی: $E_{0->} = \frac{1}{2} C V_{dd}^2$ $E_{1->} = \frac{1}{2} C V_{dd}^2$

فصل پنجم

روشهای پیاده سازی
معماری پردازشگر

چکیده

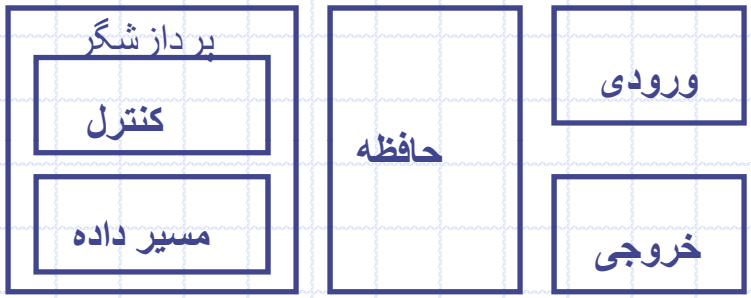
- مقدمه
- بخشهای سازنده مسیر داده
- پیاده سازی یک مسیر داده ساده (تک چرخه ای)
- پیاده سازی چند چرخه ای

مقدمه

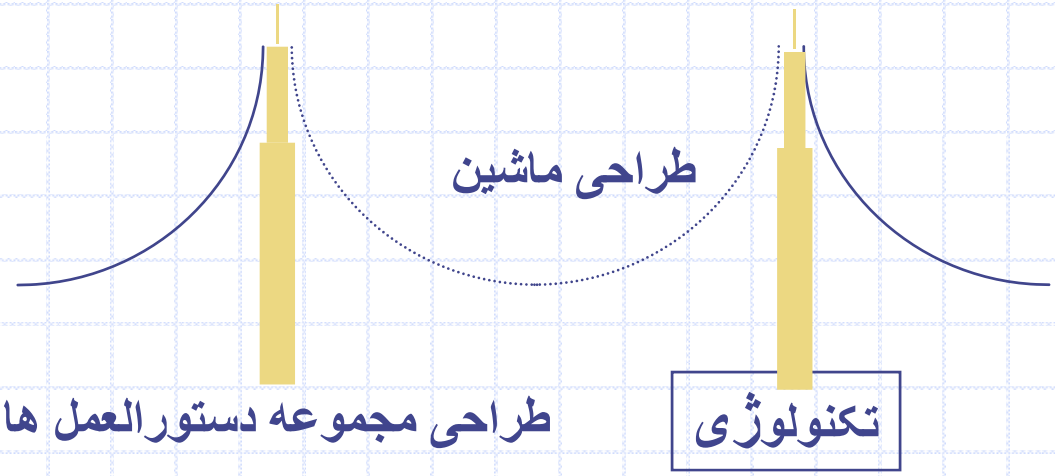
- روند پایه اجرای دستورالعملها
- برخی قراردادها
  - مسیر داده 32 بیتی
  - استراتژی ساعت زنی(حساس به لبه)
- ما به زیر مجموعه ای از MIPS تمرکز می کنیم:
  - دستورالعملهای دستیابی به حافظه:lw ، sw
  - اعمال ALU: add, sub, and, or
  - دستورات انشعاب در صورت تساوی(beq) و دستورات پرش (j)

یک تصویر بزرگ:ما الان کجا هستیم؟

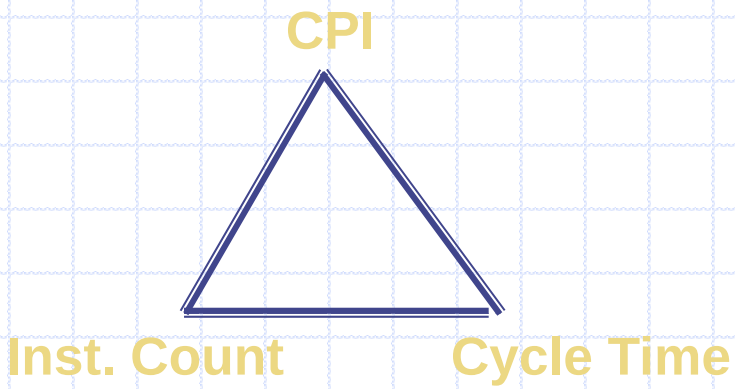
- پنج نمونه از بخشهای سازنده کامپیوتر



- موضوع روز:طراحی یک پردازشگر تک چرخه ای



یک شمای اصلی:چشم انداز کارایی



- کارایی یک ماشین تعیین می شود به وسیله:
  - تعداد دستور العمل ها(**Inst. Count**)
  - زمان هر چرخه(**Cycle Time**)
  - چرخه های هر دستورالعمل(**CPI**)
- طراحی پردازشگر (مسیر داده و کنترل) تعیین خواهد شد:
  - زمان هرچرخه
  - چرخه های هر دستورالعمل
- پردازشگر تک چرخه ای _یک کلاک دارد برای هر دستورالعمل
  - مزایا:طراحی ساده و CPI کم.
  - معایب:زمان چرخه طولانی است و این باعث می شود که به وسیله کندترین دستورالعمل محدود شود

چه طور یک پردازشگر طراحی می شود:مرحله به مرحله

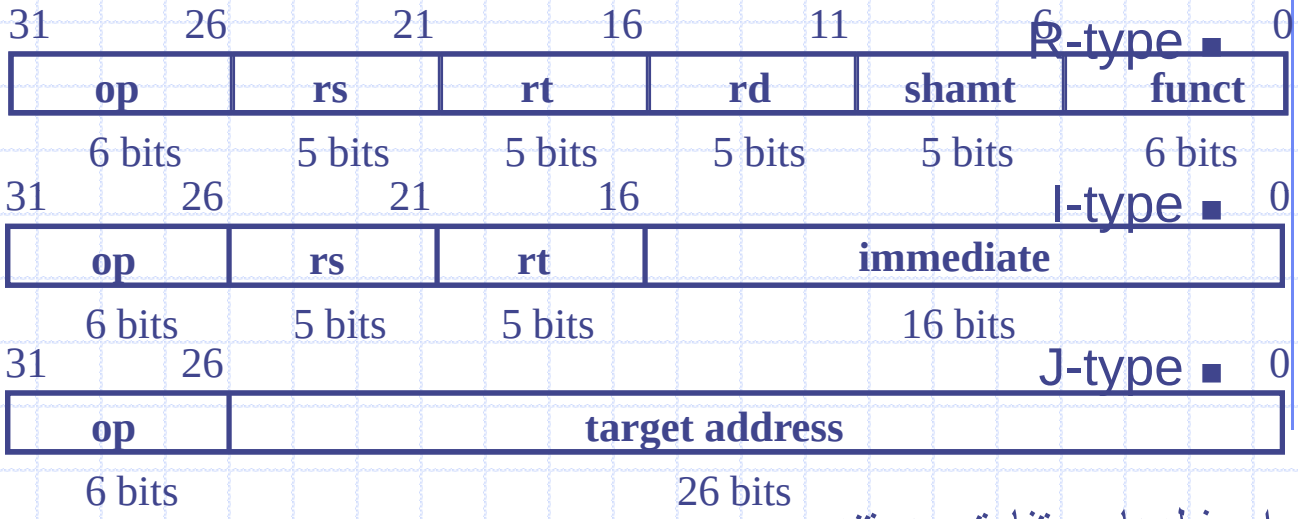
- تجزیه وتحلیل مجموعه دستورالعمل=>پیش نیازهای مسیر داده
  - مفهوم هر دستورالعملداده میشودبا انتقال دادن ثبات ها

R[rd] <– R[rs] + R[rt];

  - مسیر داده باید شامل محیط ذخیره سازی برای ثبات های ISAشود
  - مسیر داده باید انتقال دادن هر ثبات را پشتیبانی کند
- انتخاب کردن مجموعه ای از بخشهایی از مسیر داده و بنیاد نهادن روشهای ساعت زنی
- طراحی مسیر داده با در نظر گرفتن پیش نیاز ها
- Analyze implementation of each instruction to determine setting of control points that effects the register transfer
- طراحی کنترل منطقی

یادآوری:قالب های دستورالعمل های MIPS

- همه دستورالعمل های MIPSطولشان 32 بیت است. 3تا قالب دستورالعمل هستند.



دارای فیلدهای متفاوتی هستند:

- OP:عملکرد هر دستورالعمل
- Rs,rt,rd:منبع ومقصد مخصوص ثبات ها
- shamt: اندازه شییفت
- Funct:انتخاب کردن عملیات متفاوت در فیلد"op"
- address / immediate:افست آدرس یا مقدار فوری
- target address:آدرس هدف دستورالعمل پرش

ثبات منطقی انتقال (RTL)

- RTLمفهوم دستورالعمل را می دهد
- همه دستورالعمل ها با واکشی دستورالعمل آغاز می شوند

op | rs | rt | rd | shamt | funct = MEM[PC]

op | rs | rt | Imm16 = MEM[PC]

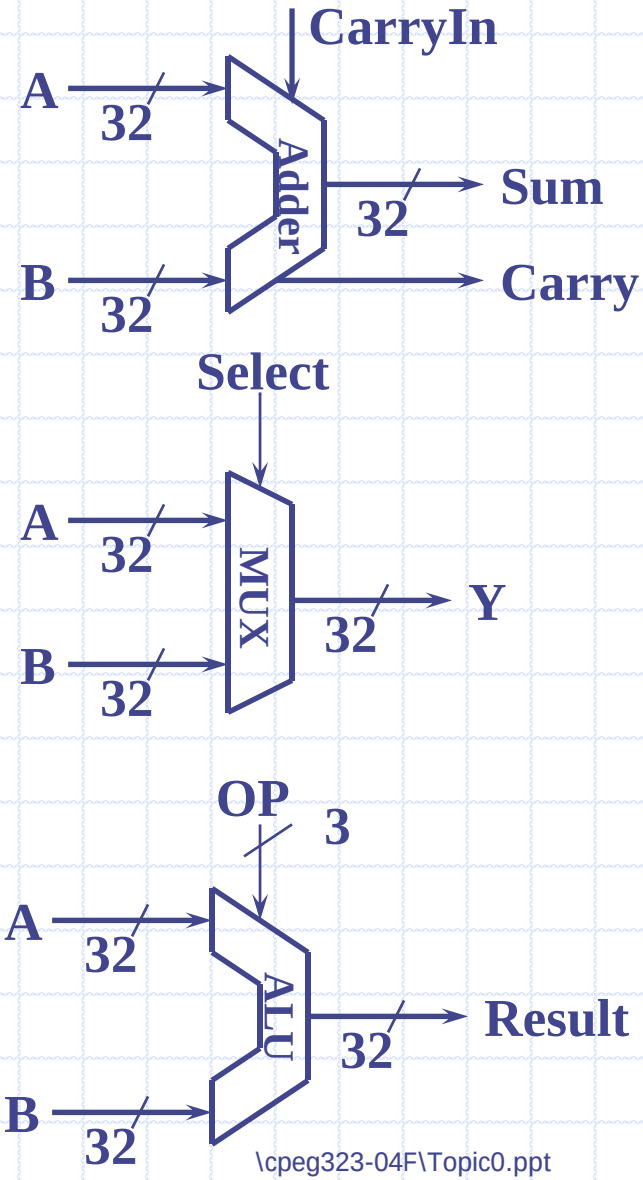
inst	Register Transfers	
addu	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
subu	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
ori	R[rt] <– R[rs] + zero_ext(imm16);	PC <– PC + 4
load	R[rt] <– MEM[R[rs] + sign_ext(imm16)];	PC <– PC + 4
store	MEM[R[rs] + sign_ext(imm16)] <– R[rt];	PC <– PC + 4
beq	if (R[rs] == R[rt]) then PC <– PC + 4 + sign_ext(imm16)] || 00 else PC <– PC + 4	

مرحله 1:تعیین پیش نیازهای مجموعه دستورالعمل

- حافظه
  - دستورالعمل و داده ها
- ثبات ها(32 x 32)
  - خواندنrs
  - خواندن rt
  - نوشتن rdیاrt
- PC شمارنده برنامه
- گسترش دهنده(گسترش علامت یا گسترش صفر)
- جمع و تفریق یا مقدار فوری گسترده شده
- Add 4 or shifted extended immediate to PC

مرحله 2: اجزاء مسیر داده

- جمع کننده
- تسهیم کننده
- واحد محاسبه ومنطق



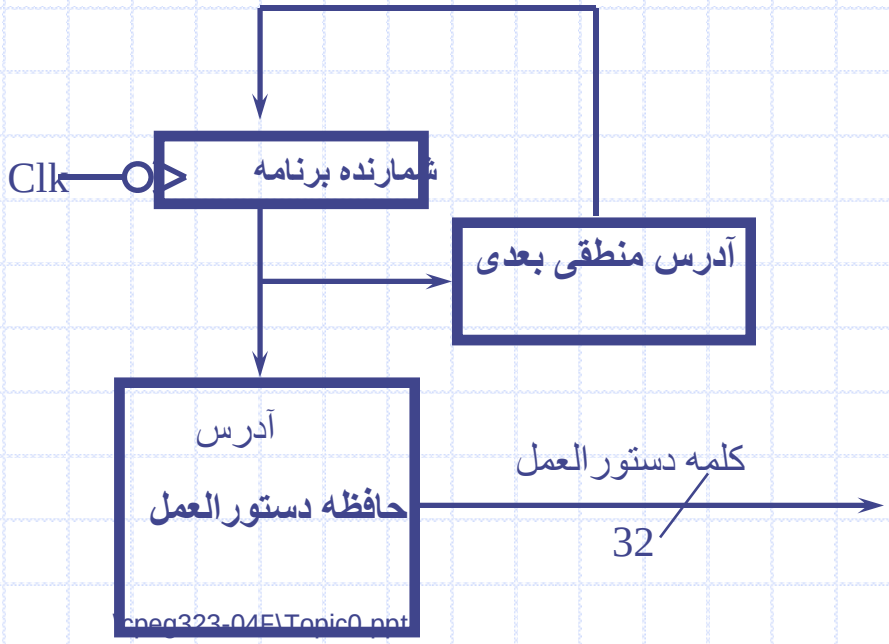
Combinational Logic:
Does not use a clock

مرحله 3

- پیش نیازهای ثبات انتقال -> طراحی مسیر داده
  - واکشی دستورالعمل
  - کدبرداری دستورالعمل ها و خواندن عملوندها
  - محاسبه کردن عمل
  - بازنویسی نتیجه

الف 3: مقدمه واحد واکشی دستورالعمل

- عملیات متداولRTL
  - واکشی دستورالعمل: mem[PC]
  - جدید کردن شمارنده برنامه:
    - کد پشت سرهم: PC <- PC + 4 :
    - انشعاب و پرش: PC <- “something else”

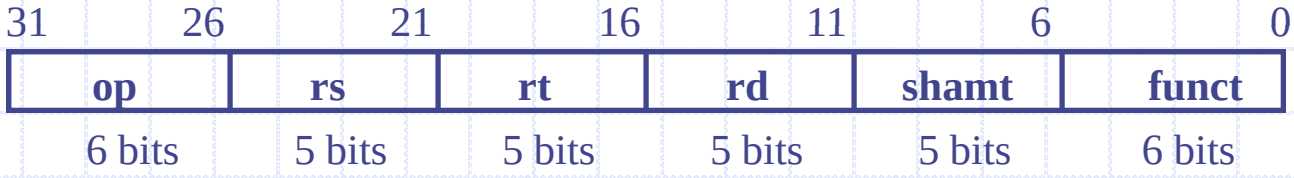


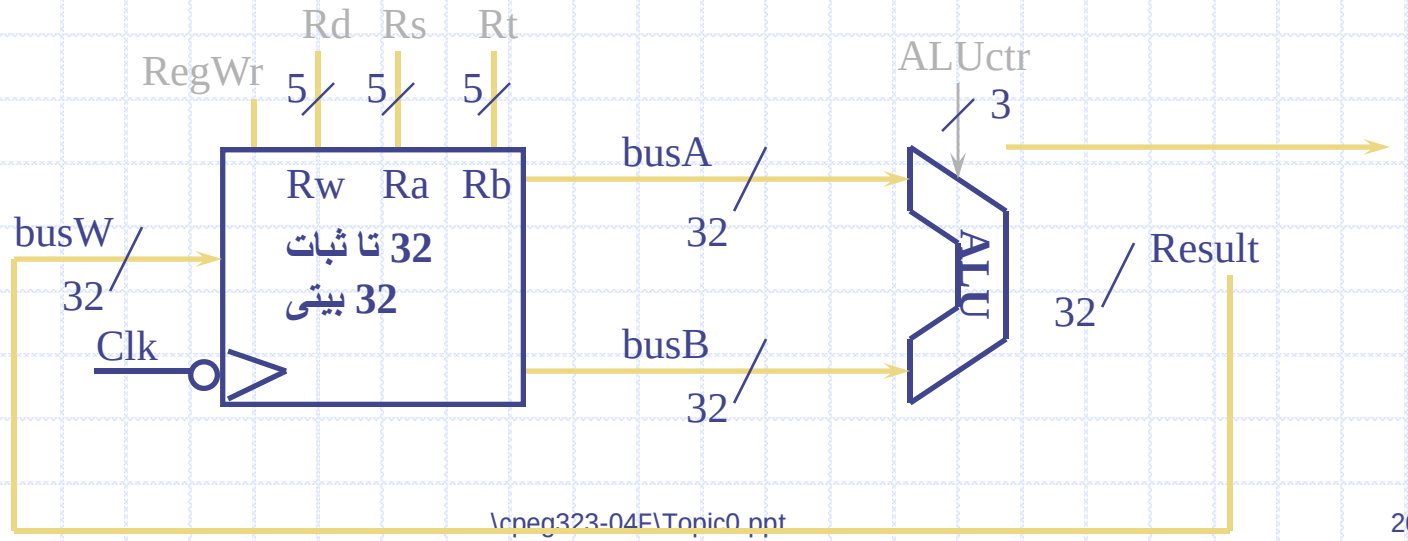
ب3:جمع و تفریق

R[rd] <- R[rs] op R[rt]

مثال: addu rd, rs, rt

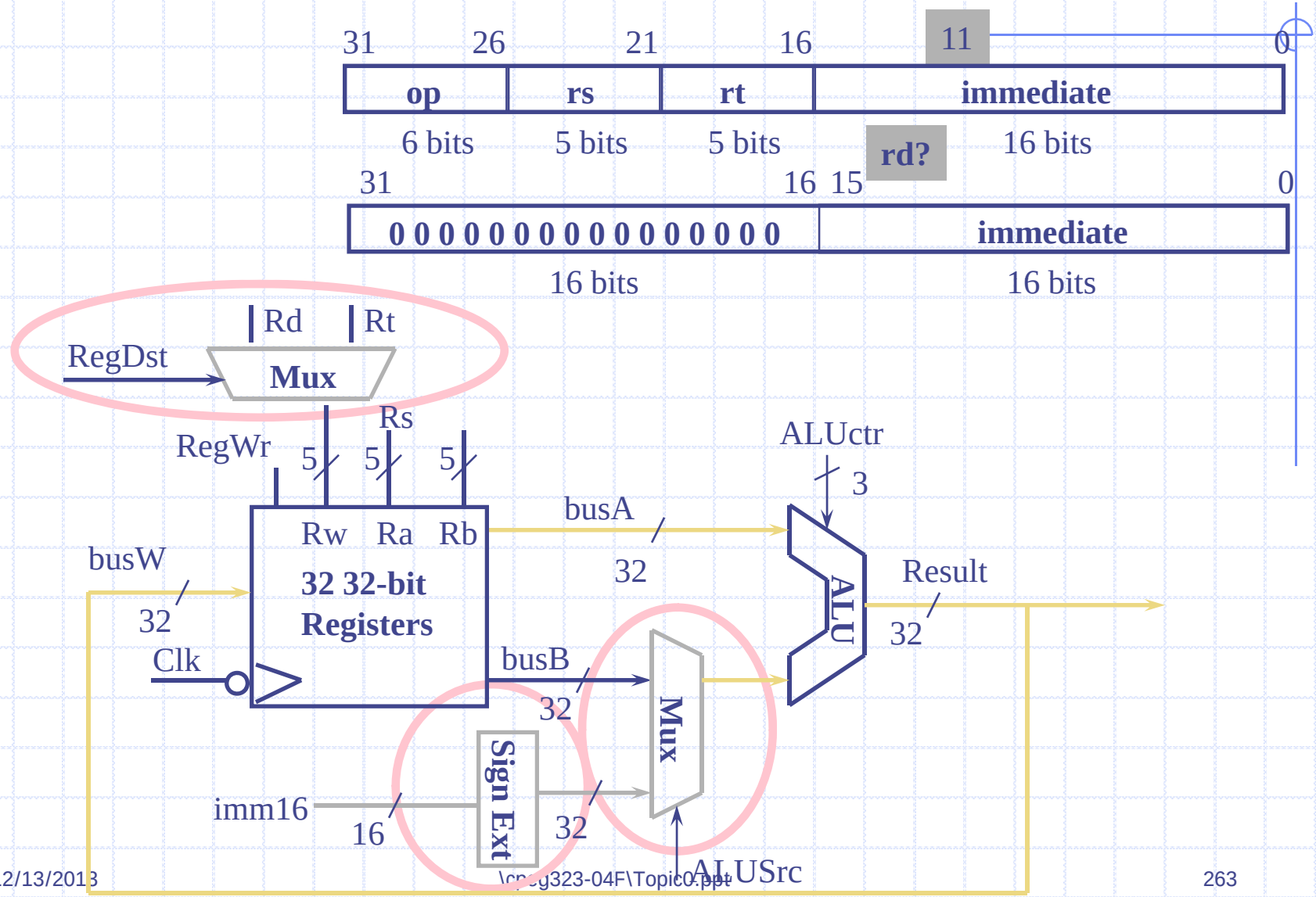
- Ra, Rb, وRw میآیند از فیلدهای دستورالعمل های rs, rt, وrd
- ALUctr and RegWr:بعد از کد برداری دستورالعمل کنترل منطقی می کنند





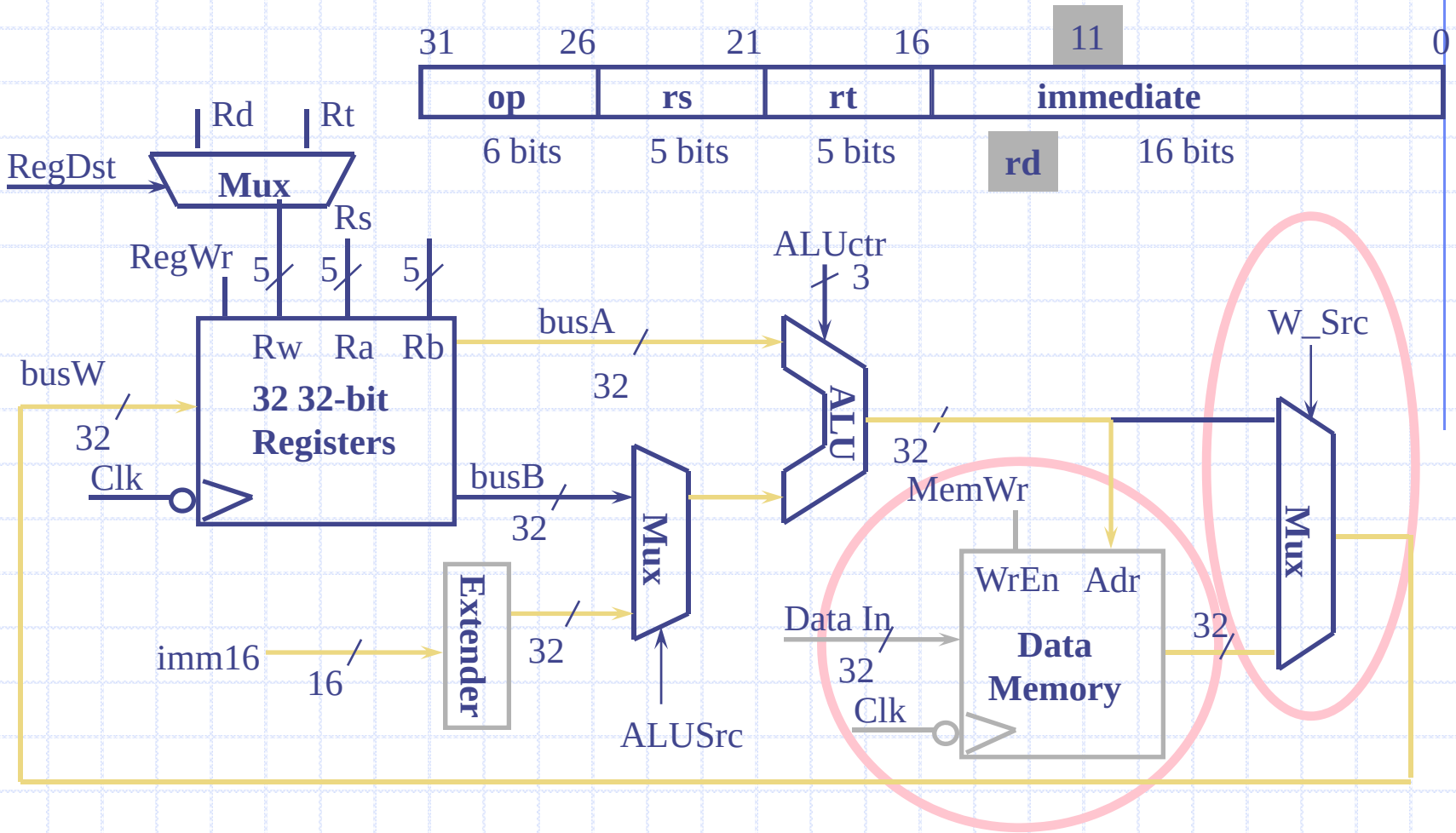
پ3:عمل منطقی بلافاصله

: ori rt, rs, imm16 مثال [imm16] گسترش صفرR[rt] <- R[rs] op



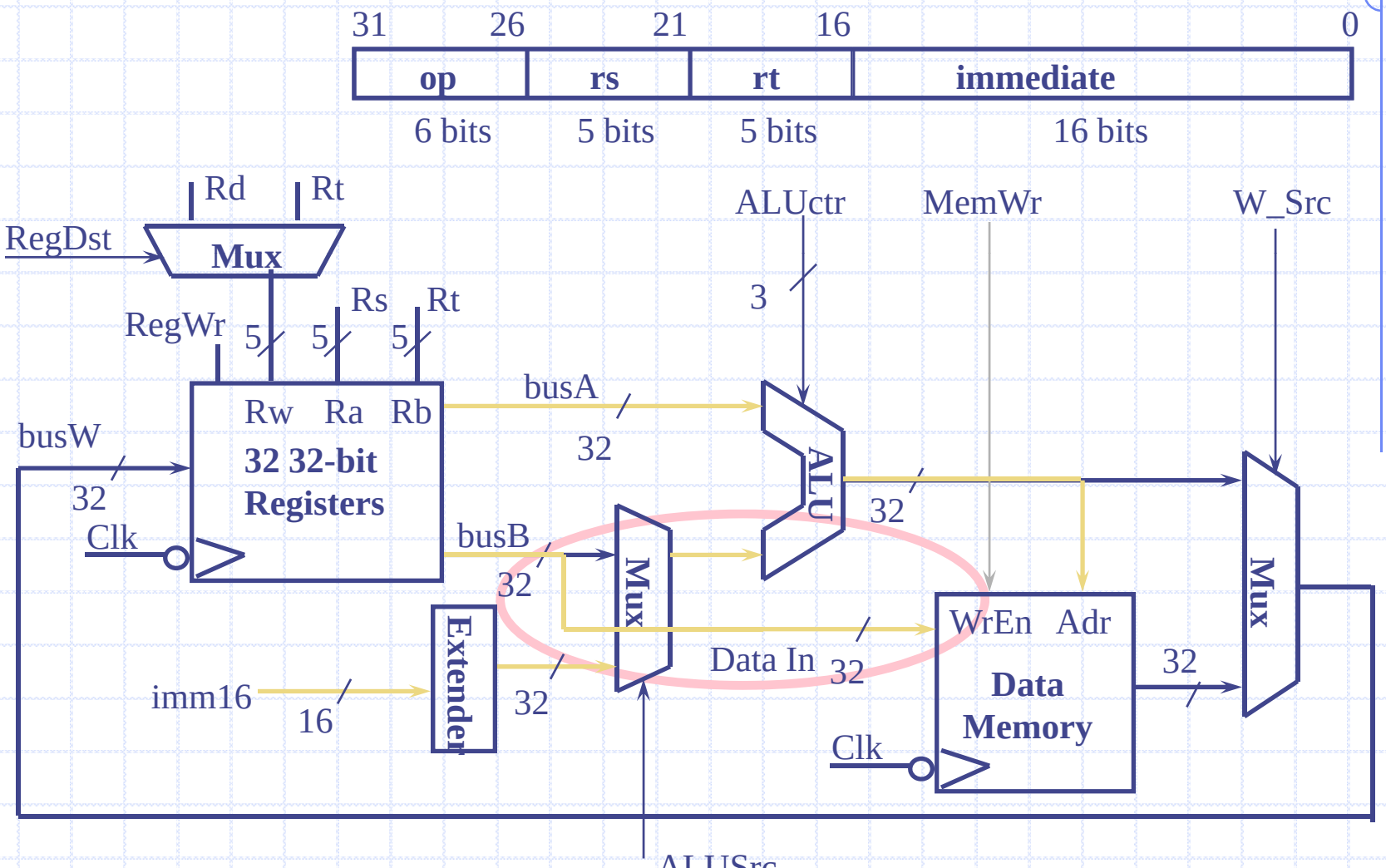
ت3:عمل بارکردن

: lw rt, rs, imm16مثال[imm16] گسترش علامتR[rt] <- Mem[R[rs] +

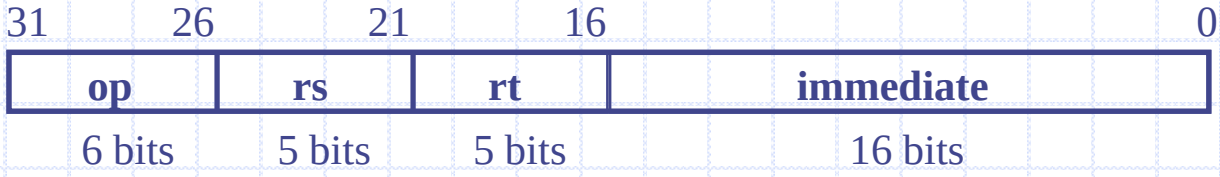


ث3:عمل ذخیره سازی

: sw rt, rs, imm16مثال[imm16] <- R[rt]] گسترش علامتMem[R[rs] +



ج3:دستورالعمل انشعاب



rs, rt, imm16 beq

واکشی دستورالعمل از حافظه mem[PC]

محاسبه شرط انشعاب Equal <- R[rs] == R[rt]

محاسبه آدرس دستورالعمل بعدی if (COND eq 0)

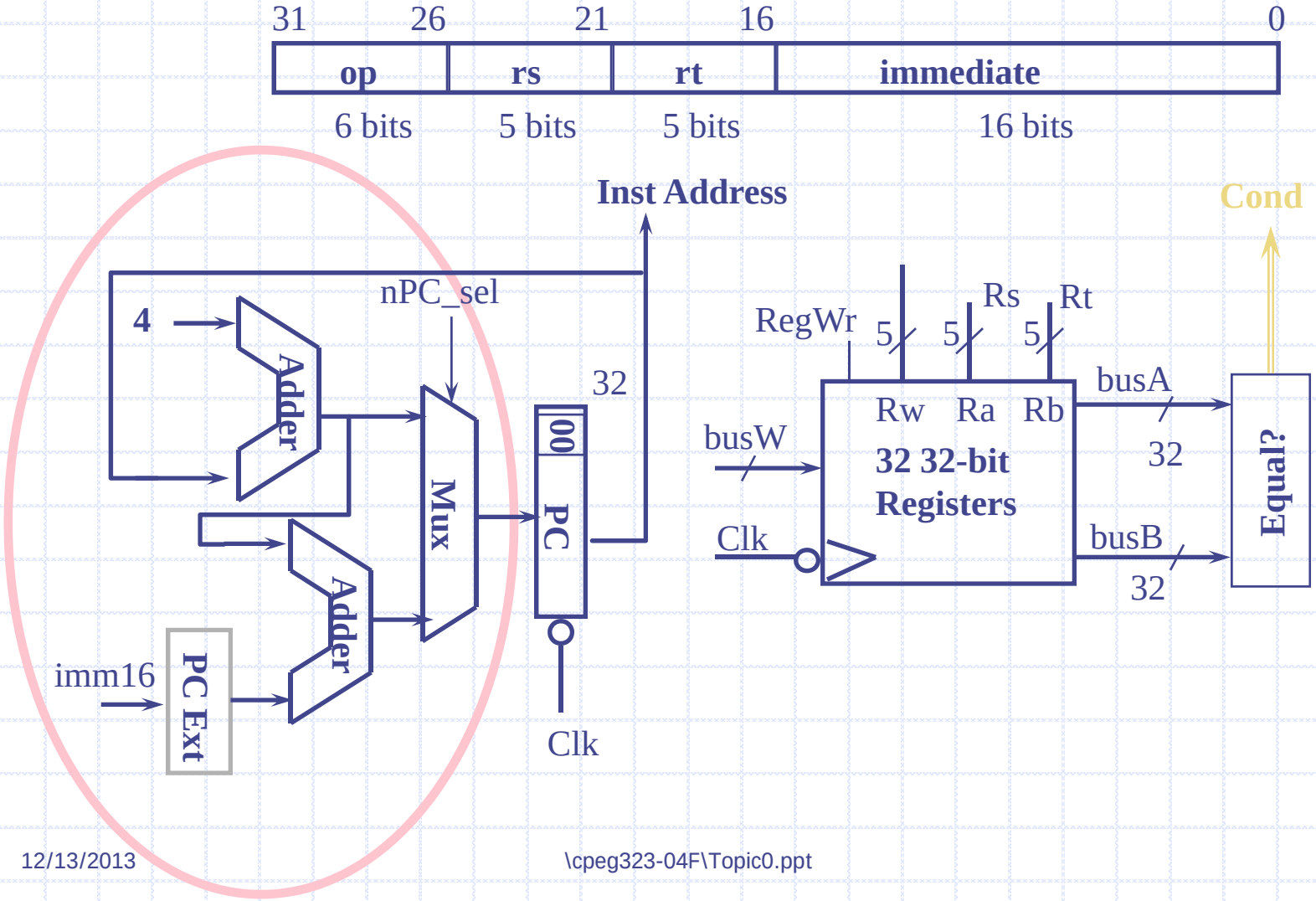
PC <- PC + 4 + (گسترش علامت(4 x (imm16)

else

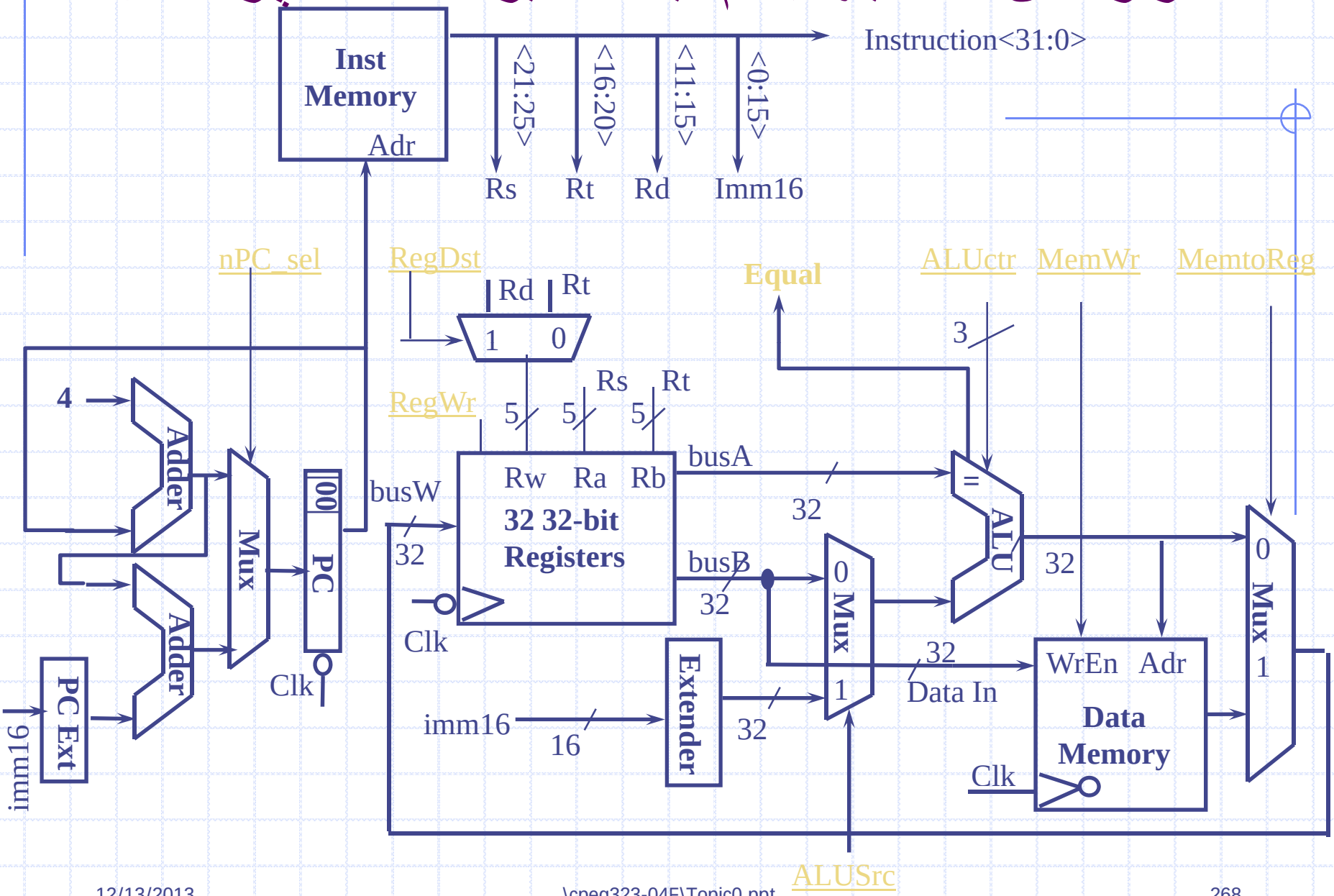
PC <- PC + 4

مسیر داده برای عمل انشعاب

ایجاد کردن مسیر داده شرطی(تساوی) beq rs, rt, imm16



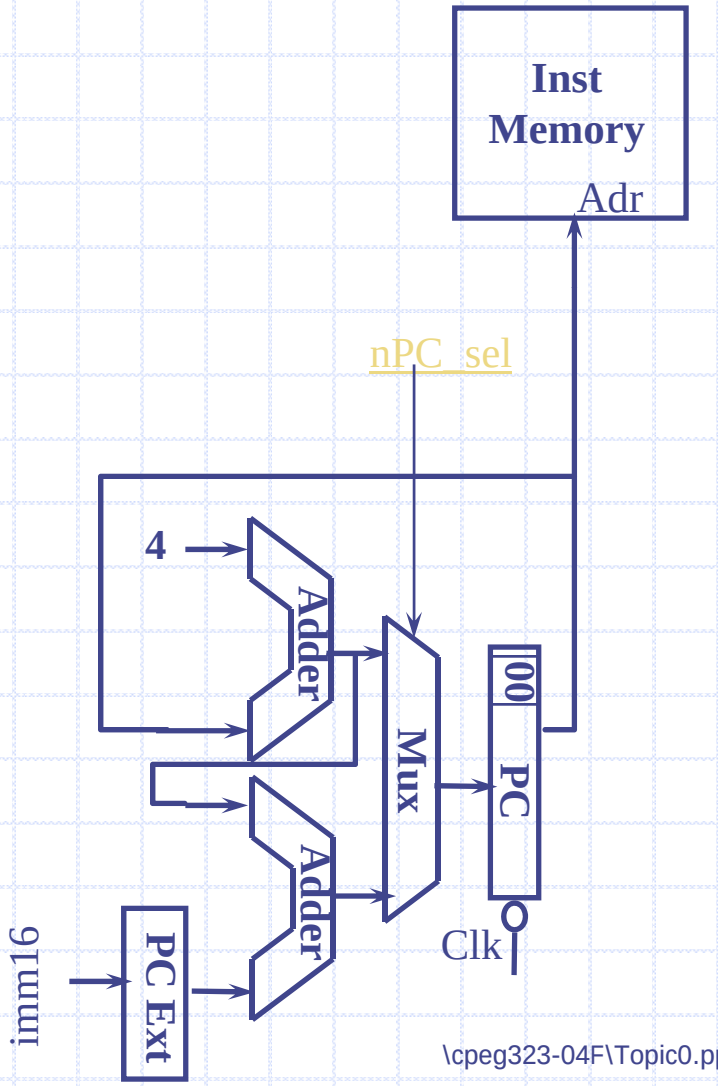
قرار دادن همه اینها باهم :یک مسیر داده تک چرخه ای



مفهوم سیگنال های کنترلی

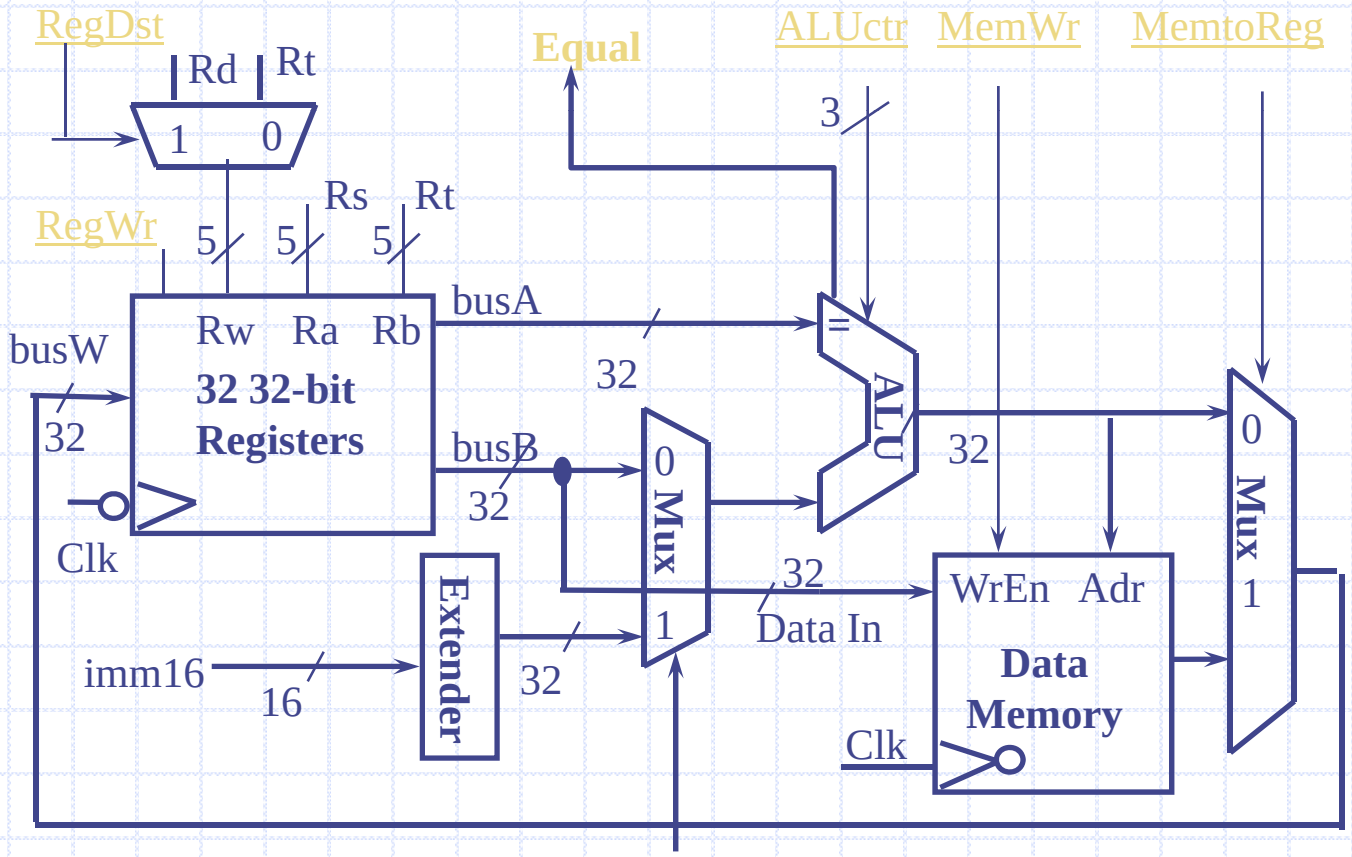
Rs, Rt, Rd وImed16 در این مسیر داده سیم بندی شده اند

nPC_sel: 0 => PC <– PC + 4; 1 => PC <– PC + 4 + SignExt(Im16) || 00



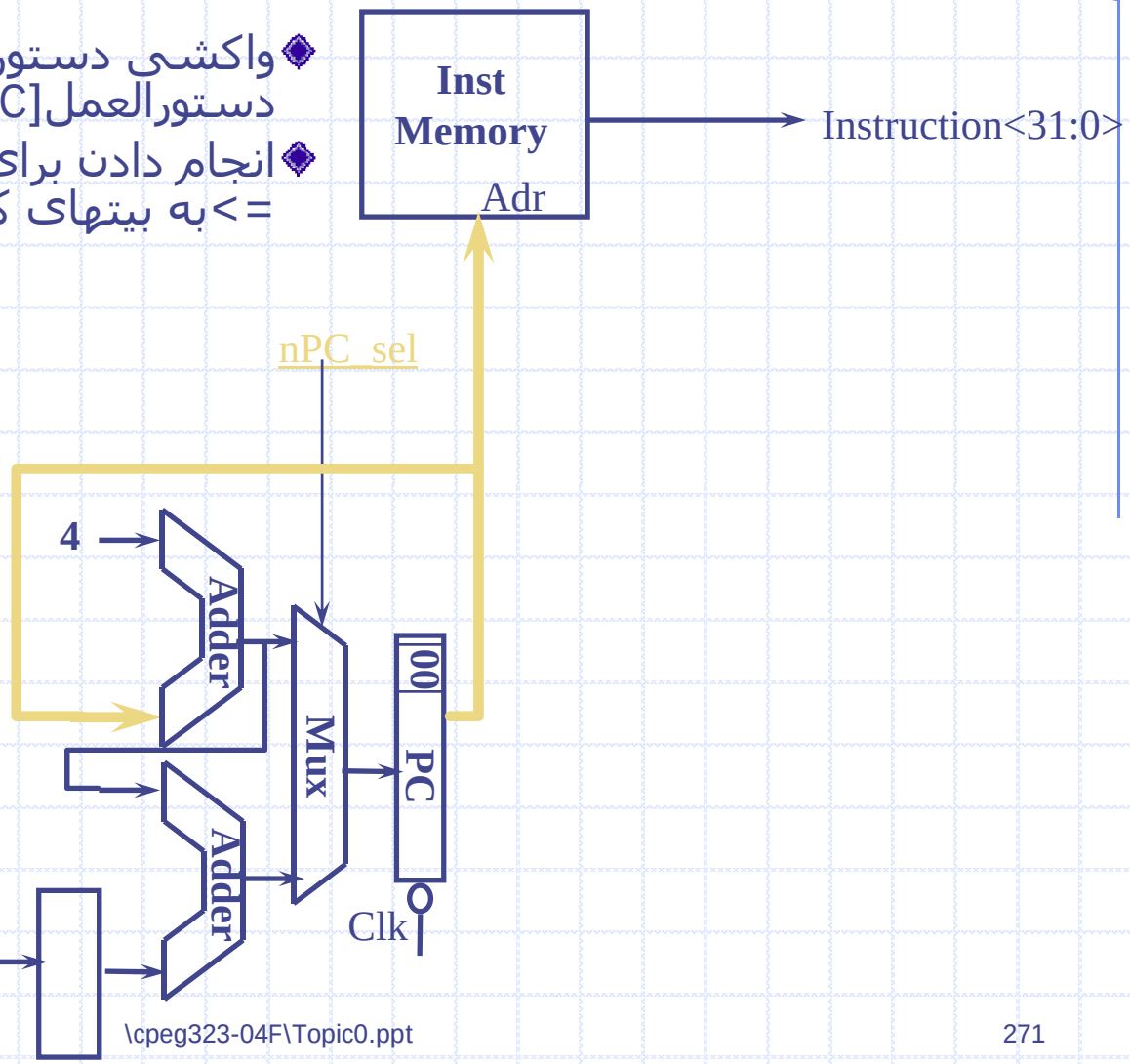
مفهوم سیگنال های کنترلی

- “zero”, “sign”ExtOp:
- 0 => regB; 1 => immedALUsrc:
- “add”, “sub”, “or”ALUctr:
- MemWr: نوشتن در حافظه
- MemtoReg: 0 => ALU; 1 => Mem
- RegDst: 0 => “rt”; 1 => “rd”
- RegWr: نوشتن در ثبات مقصد

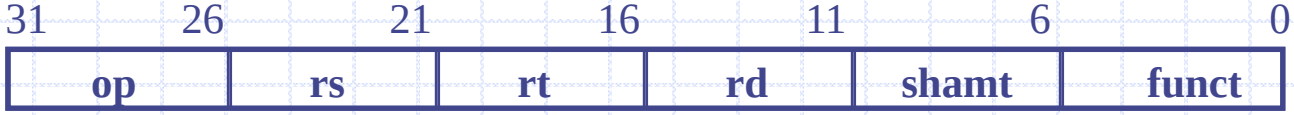


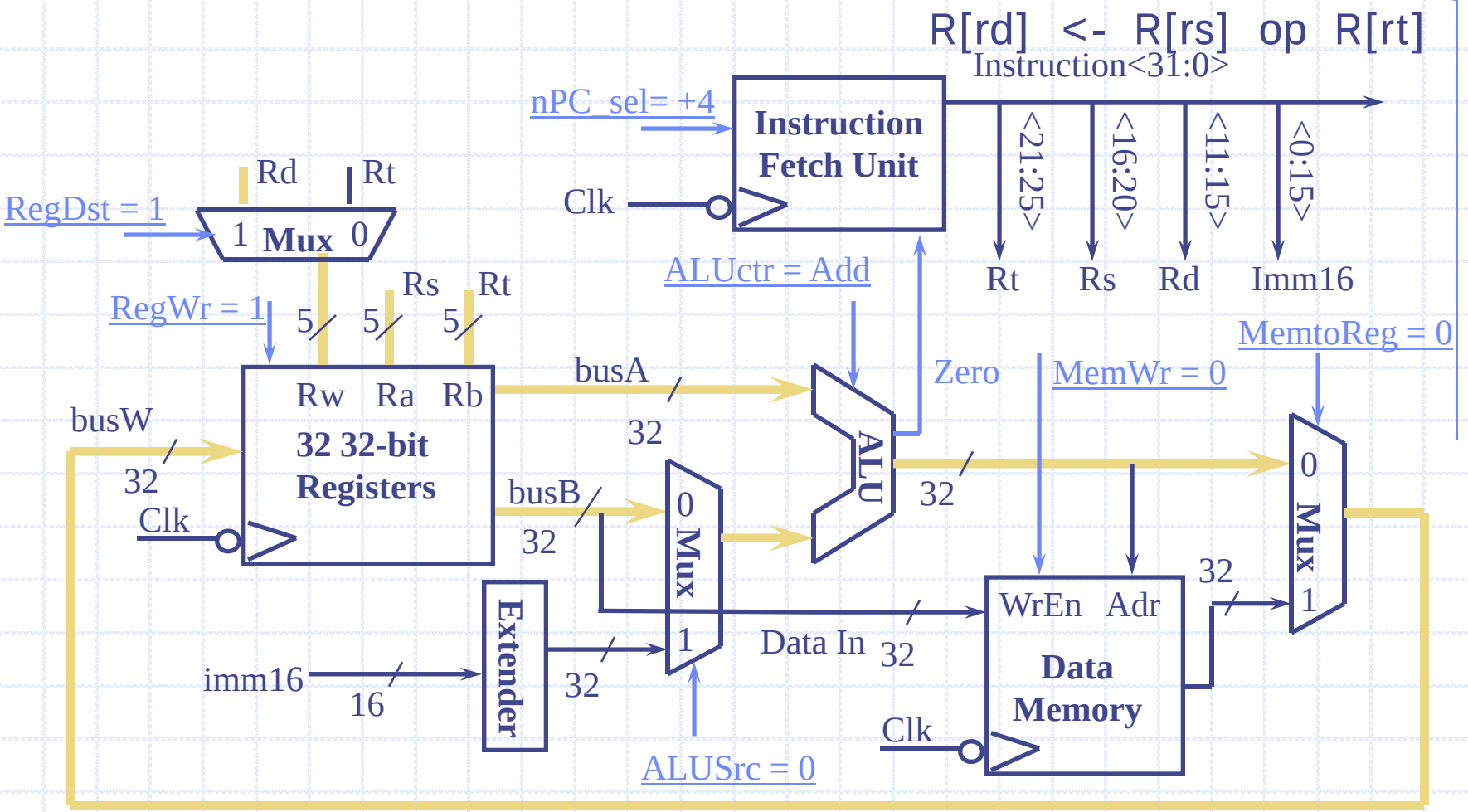
واحد واکشی دستورالعمل در ابتدای جمع

- واکشی دستورالعمل از حافظه
  instruction <- mem[PC] دستورالعمل
- انجام دادن برای همه دستورالعمل ها
  =>به بیتهای کنترلی خاصی نیاز نیست



مسیر داده تک چرخه ای در طول عمل جمع و تفریق

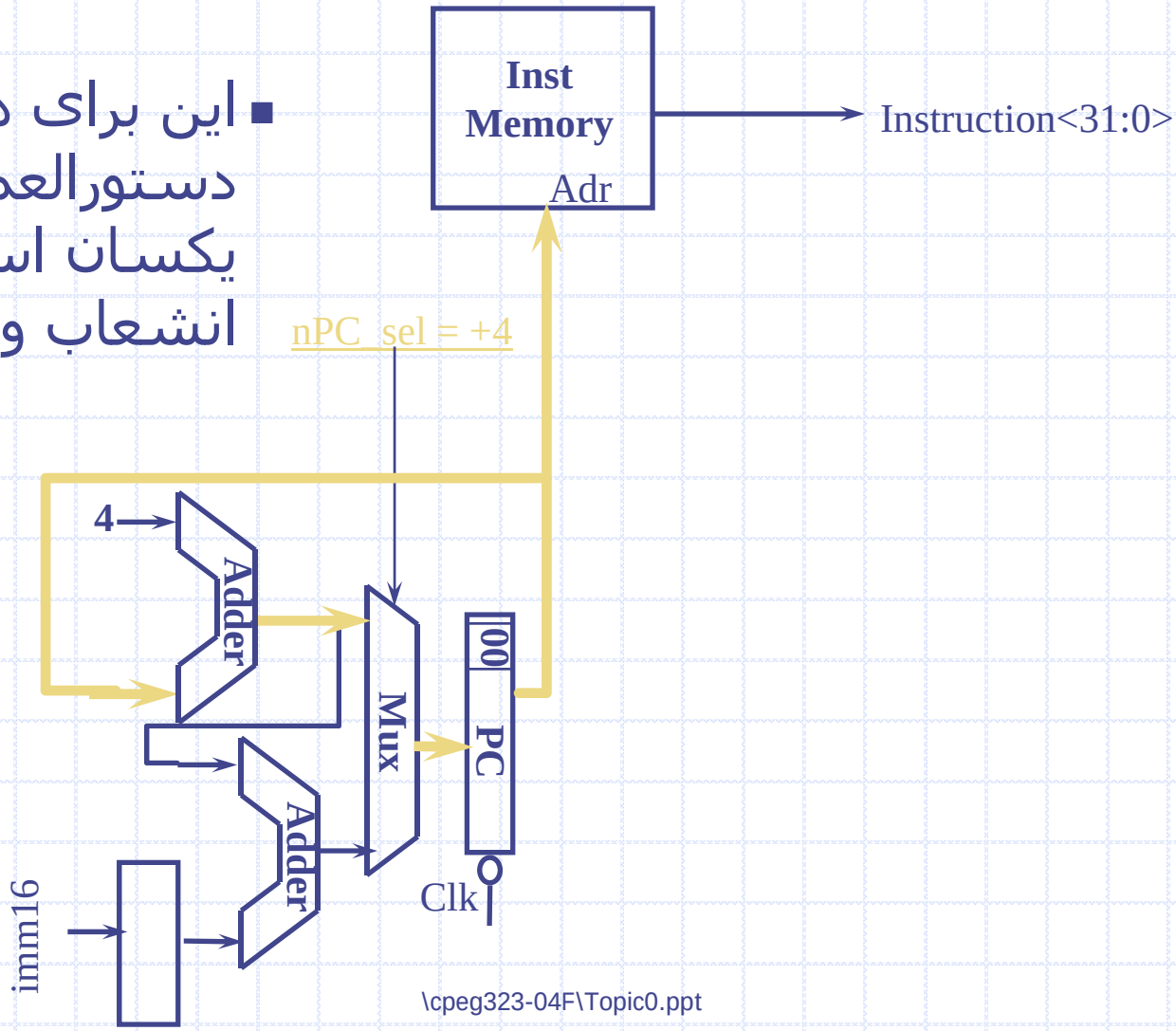




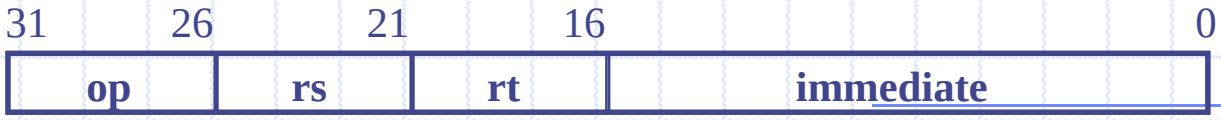
واحد واکشی دستورالعمل در انتهای جمع

PC <- PC + 4

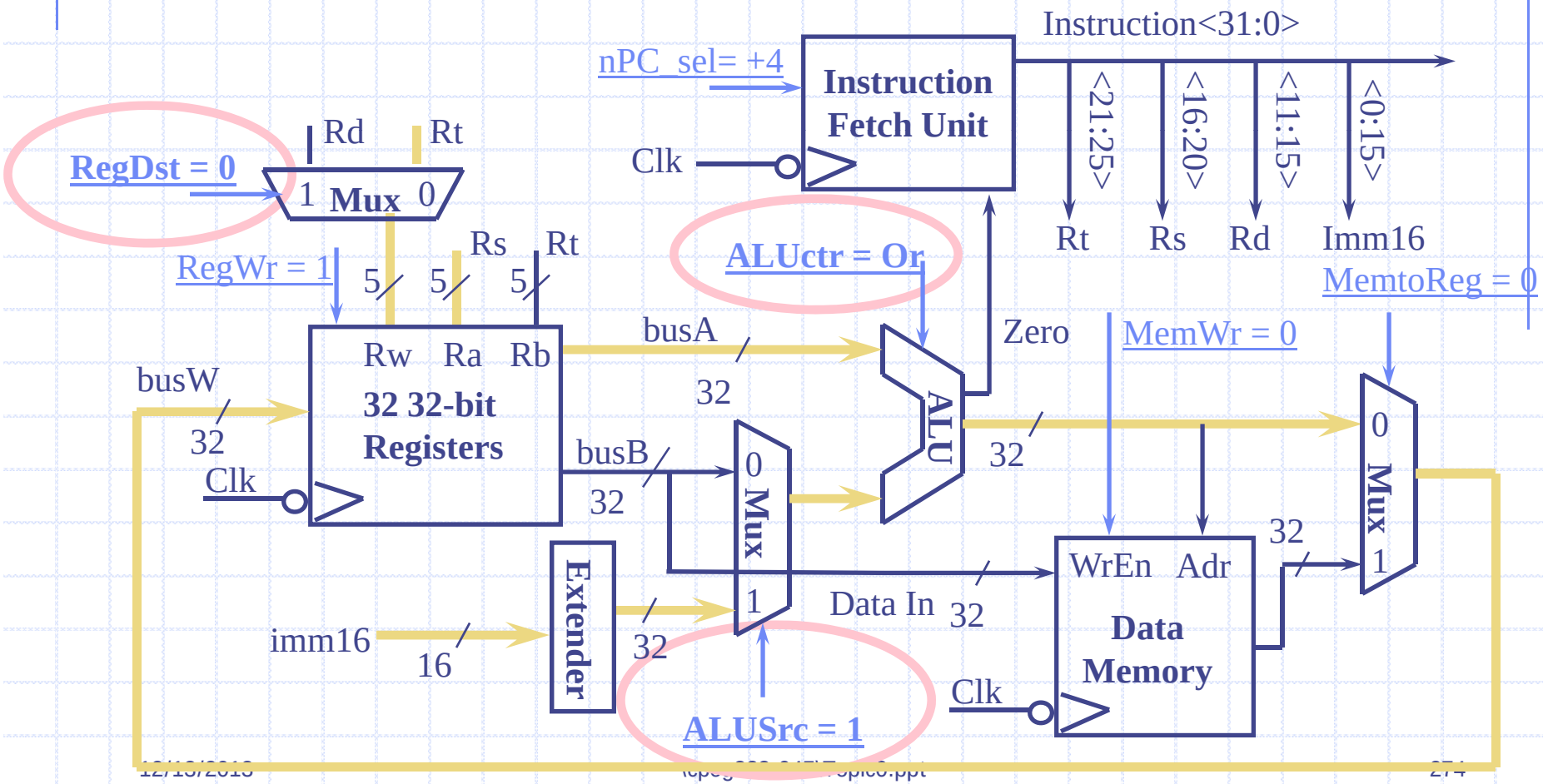
- این برای همه دستورالعمل ها یکسان است به جز انشعاب و پرش



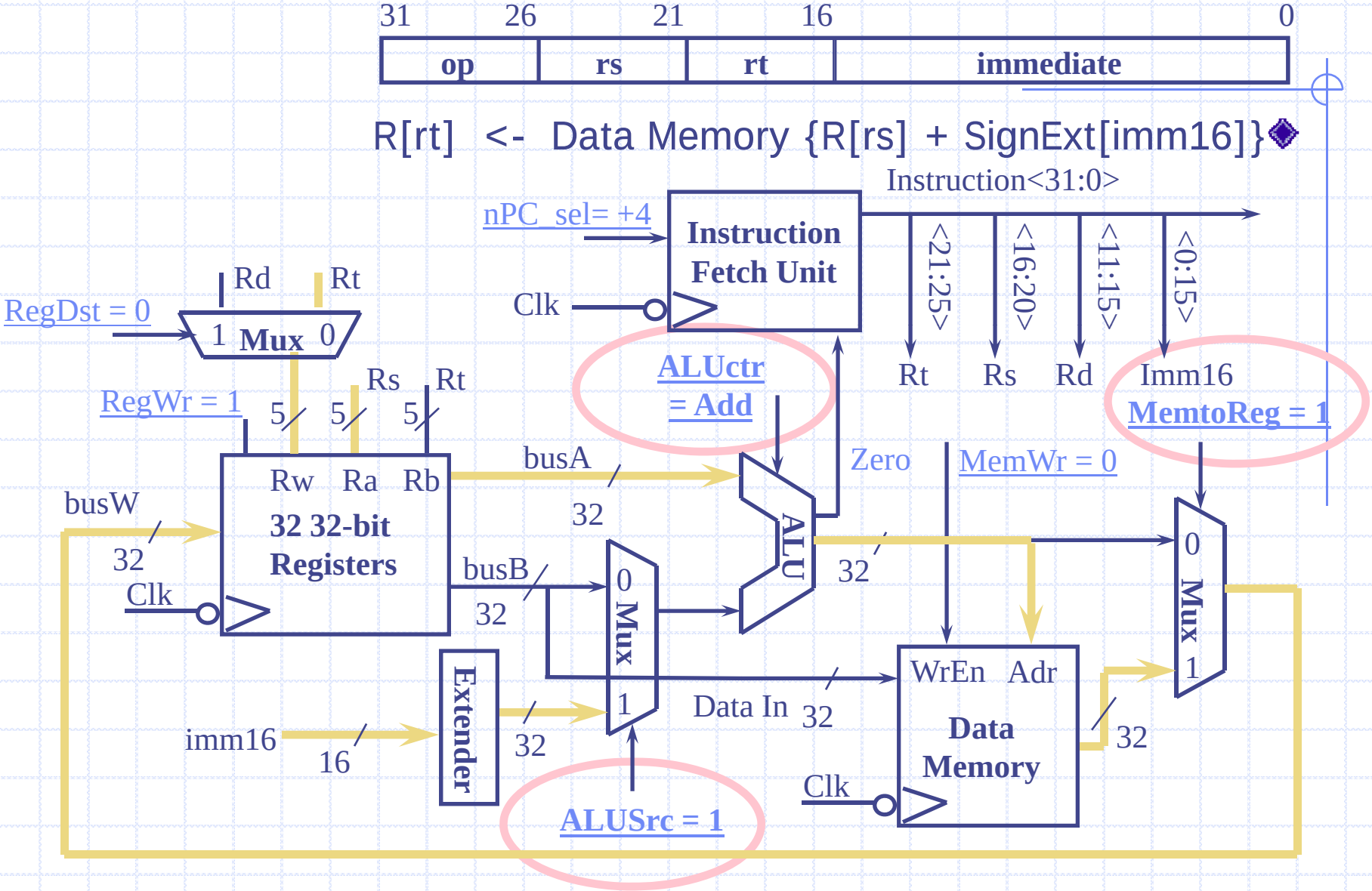
مسیر داده تک چرخه ای در طول عمل بلافاصله OR



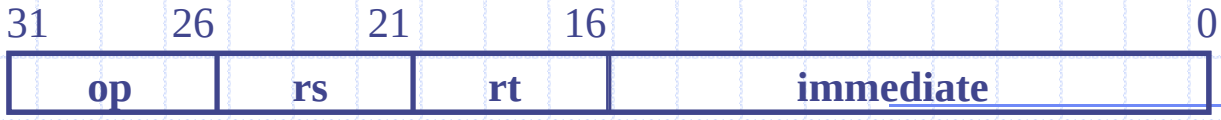
R[rt] <- R[rs] or گسترش علامت[Imm16]



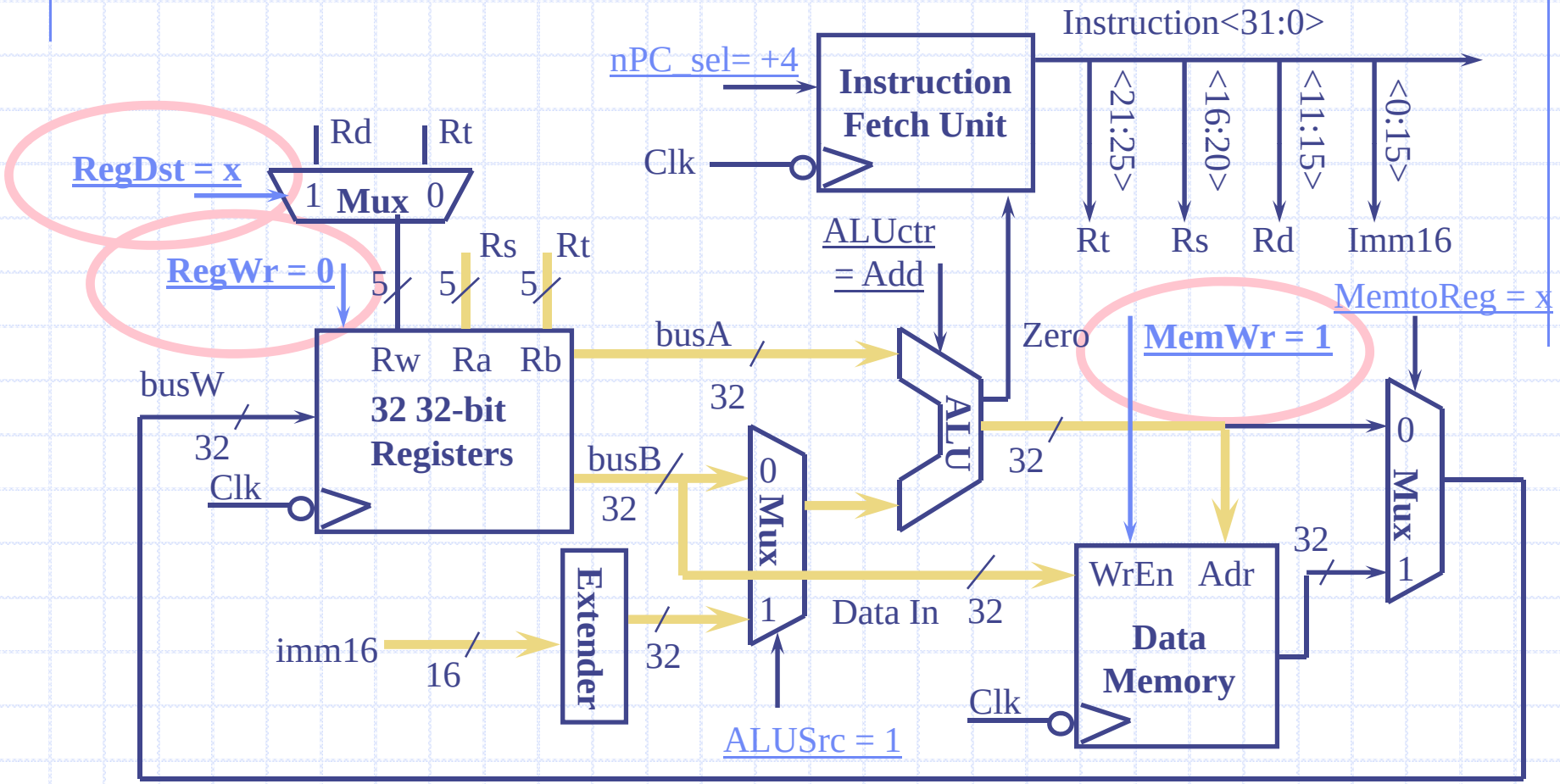
مسیر داده تک چرخه ای در طول بارکردن



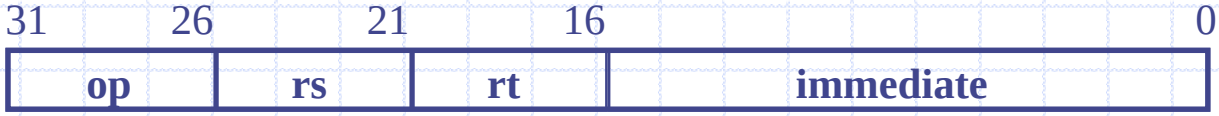
مسیر داده تک چرخه ای در طول عمل ذخیره سازی



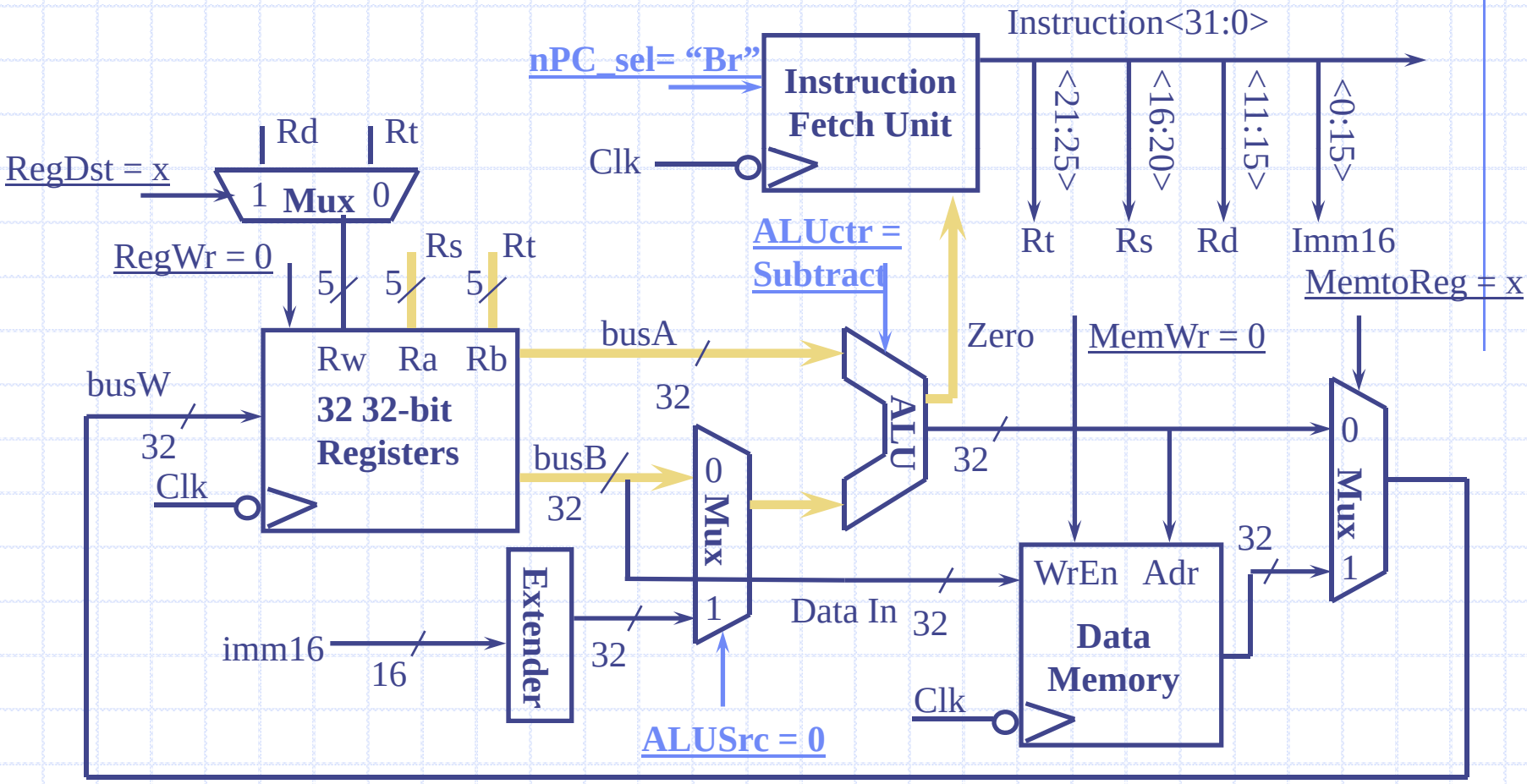
{R[rs] + SignExt[imm16]} <- R[rt] حافظه داده



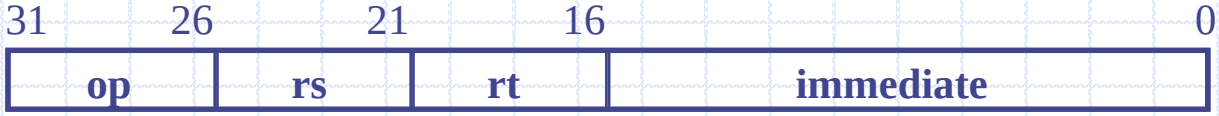
مسیر داده تک چرخه ای در طول عمل انشعاب

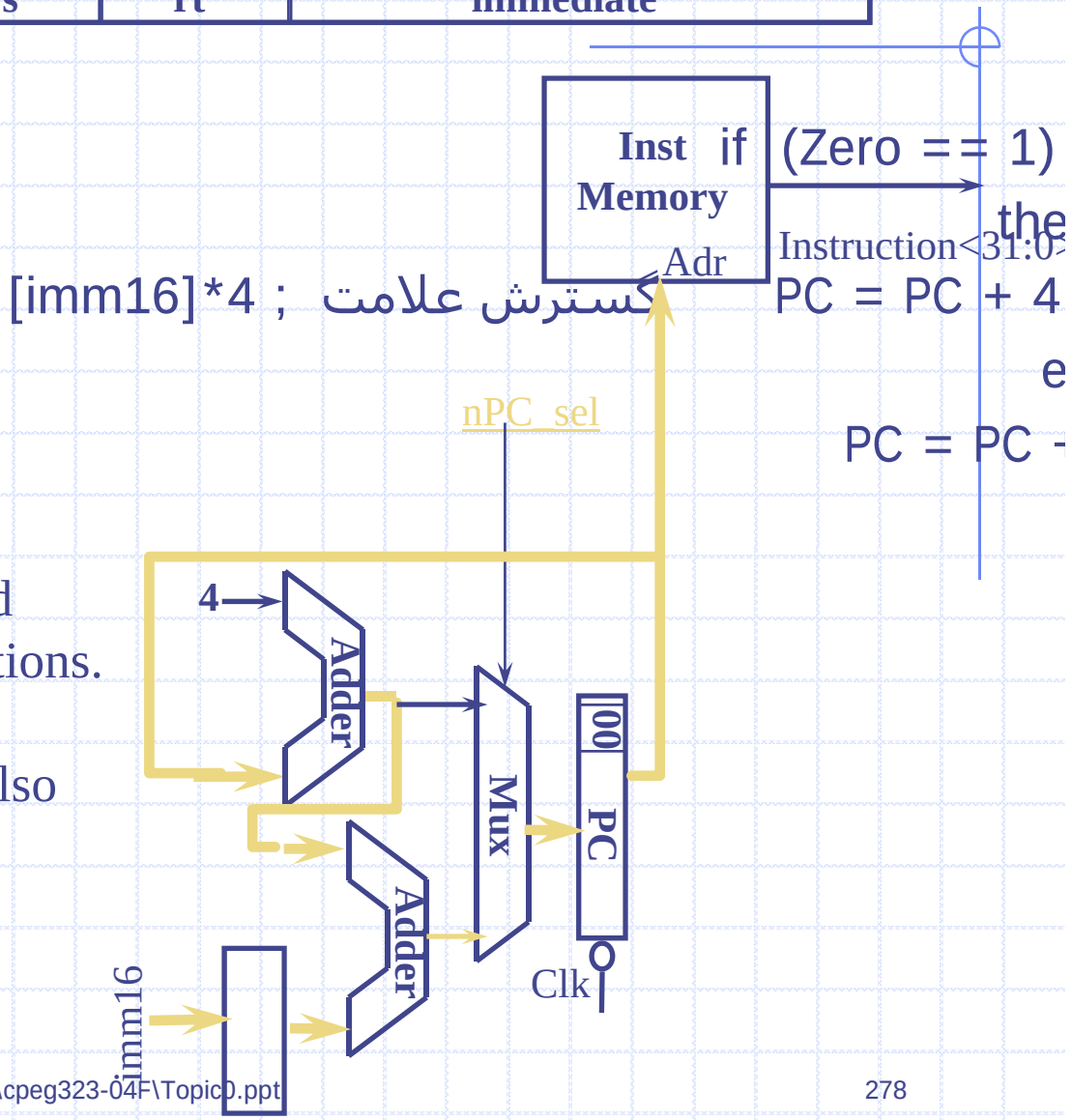


if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



واحد واکشی دستورالعمل در انتهای انشعاب

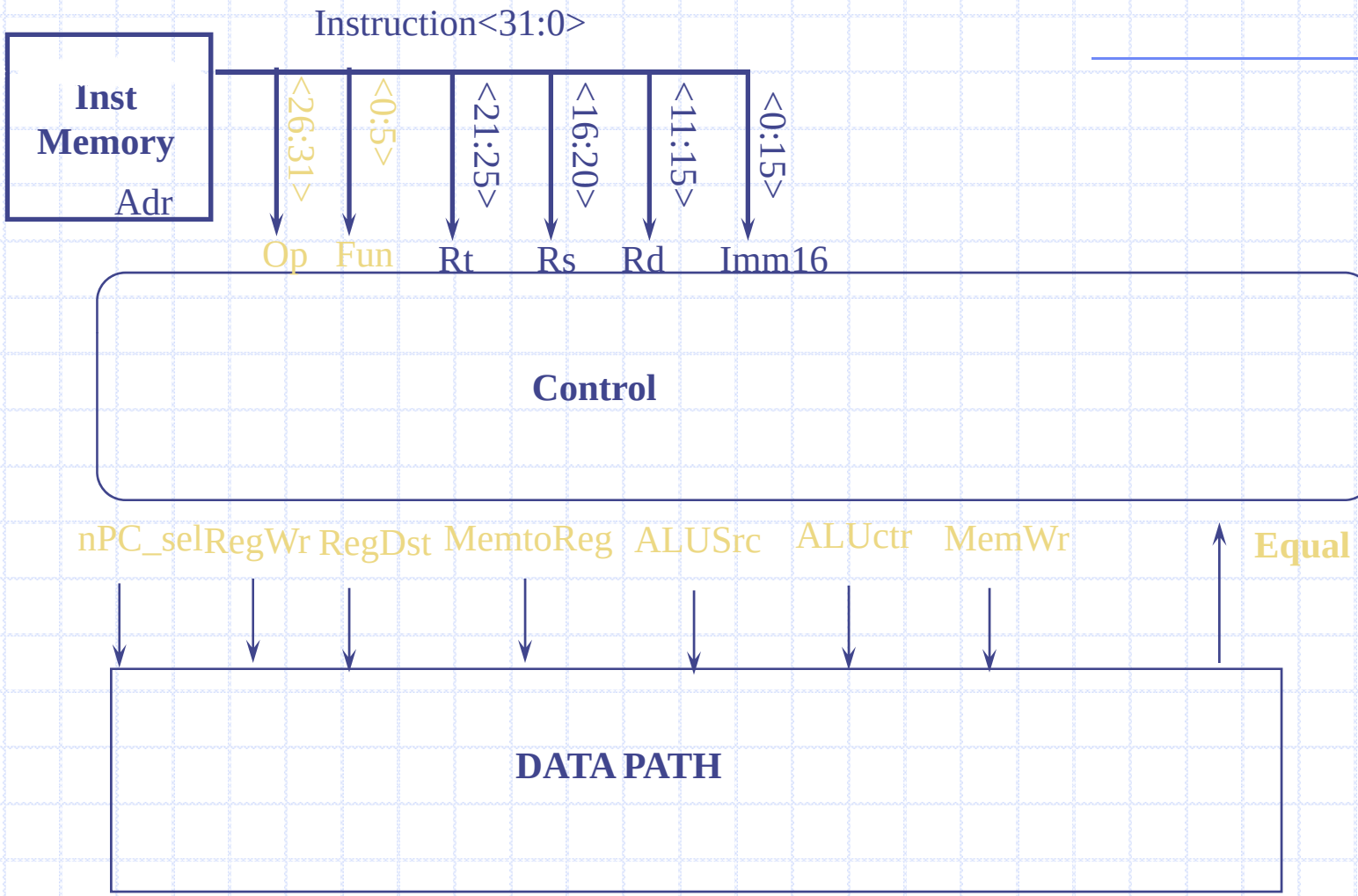




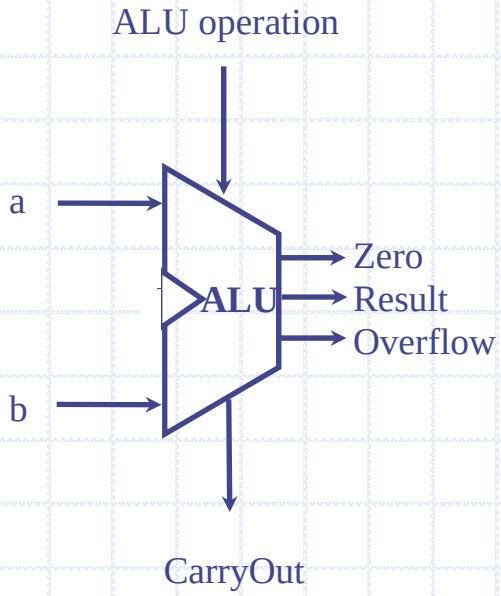
See book for what the datapath and control looks like for jump instructions.

Compared to book our processor also supports the ORI instructions.

مرحله 4:مسیر داده داده شده :RTL->CONTROL



کنترل ALU



این نمادی است که به طور معمول برای نمایش یک ALU مورد استفاده قرار می گیرد. این نماد همچنین برای نمایش یک جمع کننده نیز استفاده می شود، بنابراین بوسیله یکی از دو نماد ALU یا Adder برچسب گذاری می شود. خطوط کنترلی دارای برچسب ALUOperation می باشد. مقدار آنها و عمل ALU در شکل بعد معرفی می شوند.

کنترل ALU

ALU Control lines	Function
000	And
001	Or
010	Add
110	Subtract
111	Set-on-less-than

مقادیر سه خط کنترلی ALU به همراه عمل متناظر ALU

Instroction opcode	ALUOp	Instruction operation	Function Code	Desired ALU action	ALU control input
LW	00	Load word	XXXXXX	add	010
SW	00	store word	XXXXXX	add	010
Branch equal	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set-on-less-than	101010	set-on-less-than	111

این جدول چگونگی تنظیم بیتهای کنترلی ALU وابسته به بیتهای کنترلی ALUop و کدهای تابعی متفاوت برای دستورات نوع ثباتی را نشان می دهد.

Opcode، که در اولین ستون لیست شده است،وضعیت بیتهای ALUop ار تعیین می کند. تمام کد گذاریها به صورت دودویی نمایش داده می شوند. توجه کنید زمانی که کد ALUop برابر 00 یا 01 باشد، فیلد های خروجی به فیلد کدهای تابعی وابسته نیست. در این مورد، گفته می شود که مقدار کدهای تابعی برای ما "بدون توجه" (don't care) می باشد و فیلد کد تابعی به صورت xxxxx نمایش داده می شود. زمانی که مقدار ALUop برابر 10 باشد، کد تابعی برای تنظیم ورودی کنترل ALU به کار می رود.

دستورالعملهای که از ALU استفاده می کنند

Load/store:	محاسبه آدرس - جمع
Branch eq:	تفریق
R-type:	جمع / تفریق and/or set-on-less-than

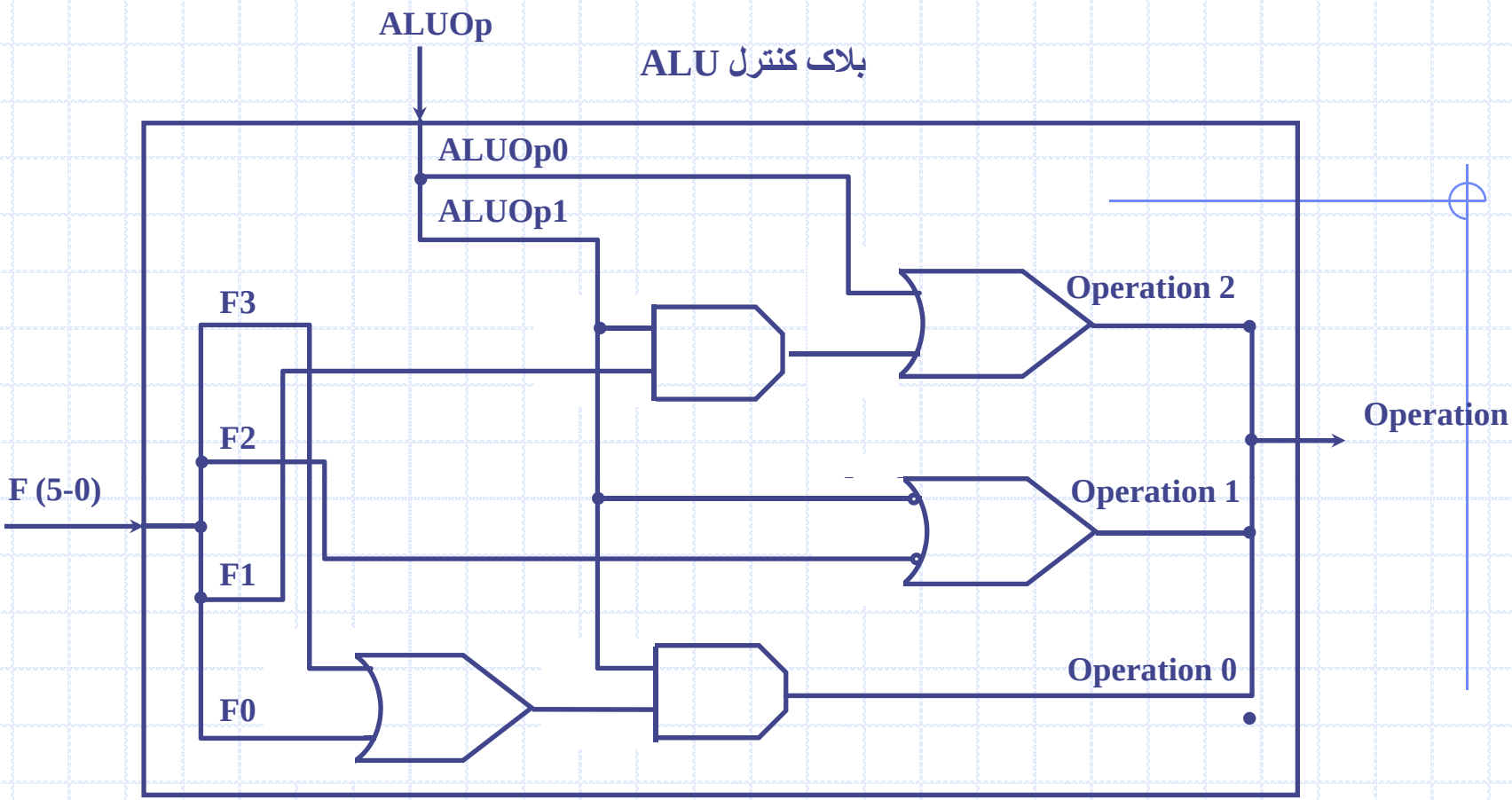
نیاز به کدهای تابعی

طراحی کنترل ALU

ALUOp			Function code					ALU control input
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

جدول درستی برای سه بیت کنترلی ALU به عنوان تابعی ازALUop و فیلد کد تابعی

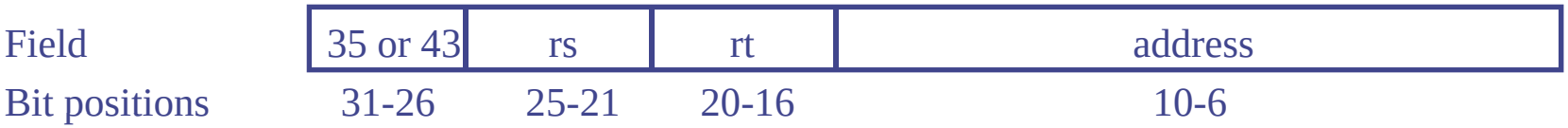
بسیاری از ورودیهای اضافه شده "بدون توجه"(don't care) هستند. به عنوان مثال، ALUop از کد گذاری 11 استفاده ای نمی کند، بنابراین جدول درستی می تواند ترجیحاً شامل ورودیهای 1x و x1 به جای 10 و 01 باشد.



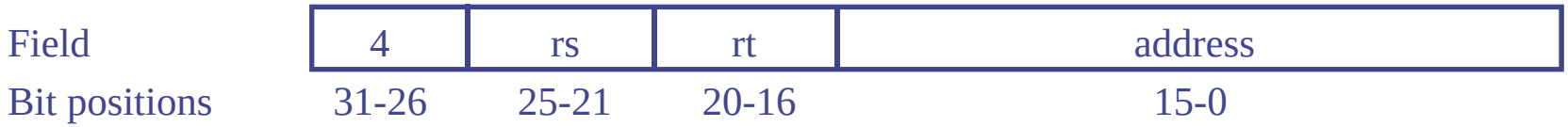
بلاک کنترلی ALU سه بیت کنترلی ALU را مبنی بر کد تابعی و بیتهای ALUop تولید می کند.

چهره قالب دستورالعملهای MIPS

a. R-type instruction



b. Load or store instruction

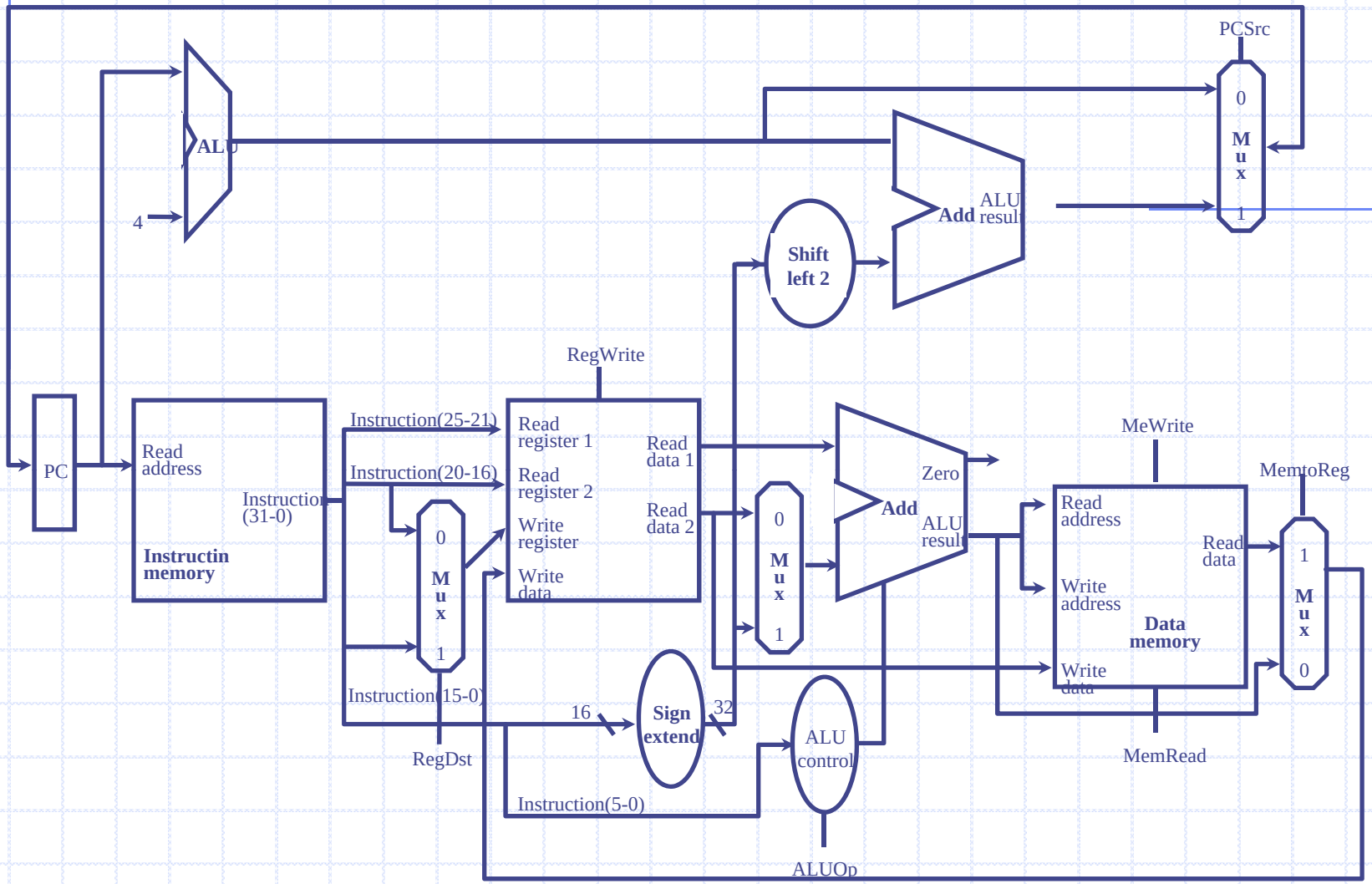


c. branch instruction

چهره قالب دستورالعملهای MIPS

cont'd

- فیلد op ، همچنین معروف به opcode، همیشه در بیتهای 26 تا 31 قرار می گیرد. ما به این فیلد به صورت Op[5-0] اشاره می کنیم.
- دو رجیستر خواندن همیشه با فیلدهای rs و rt در موقعیت 21 تا 25 و 16 تا 20 مشخص می شوند. این مطلب برای دستورات ثباتی، انشعاب در صورت تساوی و ذخیره سازی صدق می کند.
- ثبات پایه برای دستورات load و store همیشه در موقعیتهای بیتی 21 تا 25 (rs) قرار دارد.
- 16 بیت افست برای دستورات انشعاب در صورت تساوی، load و store همیشه در موقعیتهای 0 تا 15 قرار دارد.
- ثبات مقصد در یکی از این دو مکان است. برای دستورات load در بیتهای موقعیت 16 تا 20(rt) و در حالی که برای یک دستورالعمل ثباتی در موقعیتهای بیتی 11 تا 15(rt) است. بنابراین ما به افزودن یک مالتی پلکسر برای انتخاب اینکه کدام فیلد دستورالعمل به شماره رجیستر نوشتن اشاره می کند.



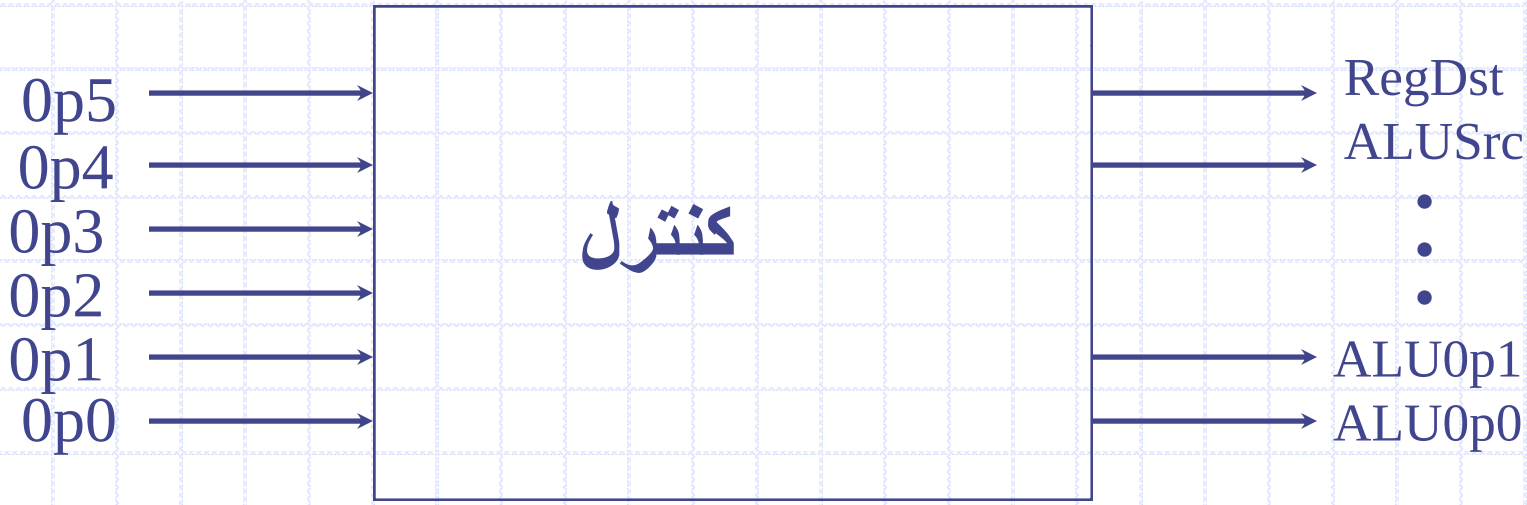
مسیر داده به همراه تمامی مالتی پلکسرها و تمامی خطوط کنترلی شناخته شده

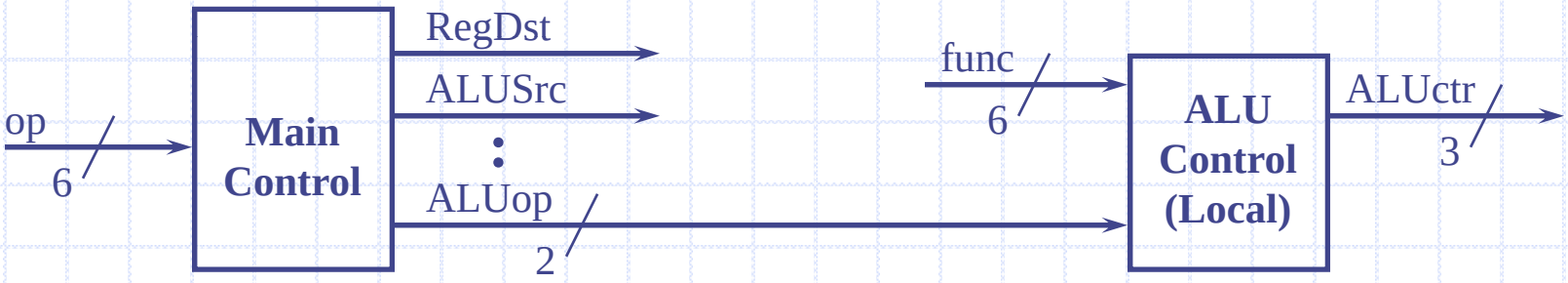
طراحی واحد کنترل برای اجرای دستورالعمل تک چرخه ای

Signal name	Effect when deasserted	Effect when asserted
MemRead	None	Data memory contents at the read address are put on read data output
MemWrite	None	Data memory contents at address given by write address is replaced by value on write data input.
ALUSrc	The second ALU operand comes from the second Register file output.	The second ALU operand is the sign-extended lower 16-bits of the instruction.
RegDst	The register destination number for the Write register comes from the rt field	The register desination number for the Write register comes from the rd field.
RegWrite	None	The register on the Write register input is written into with the value on the write data input.
PCSrc	The PC is replaced by the output of the adder That computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemtoReg	The value fed to the register write data input comes from the ALU	The value fed to the register write data input comes from the data memory.

عمل هر کدام از هفت سیگنال کنترلی. زمانیکه بیت کنترل (1) به سمت مالتی پلکسر هدایت شود، مالتی پلکسر ورودی متناظر با پایه 1 را انتخاب می کند. در غیر اینصورت در صورتی که بیت کنترل آزاد نگردد (0 باشد) مالتی پلکسر ورودی مربوط به پایه 0 را انتخاب می کند. به خاطر بسپارید که همه عناصر حالت دارای یک ورودی ساعت به عنوان یک ورودی ضمنی می باشد و ساعت به عنوان کنترل کننده نوشتن ها به کار می رود.

طراحی واحد کنترل




op
6
Main Control
RegDst
ALUSrc
ALUop
2
func
6
ALU Control (Local)
ALUctr
3

	R-format		lw	sw	beq
Inputs	0p5	0	1	1	0
	0p4	0	0	0	0
	0p3	0	0	1	0
	0p2	0	0	0	1
	0p1	0	1	1	0
	0p0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUPp0	0	0	0	1

عملکرد کنترل برای پیاده سازی تک چرخه ای ساده کاملاً به وسیله جدول درستی تعیین می شود. نیمه بالای جدول ترکیب سیگنالهای ورودی که مرتبط با 4 کد عملیاتی (opcode) می باشد و تعیین کننده وضع خروجی کنترل هستند را به دست می دهد. (به خاطر داشته باشید که Op(0-5) مرتبط با بیتهای 26 تا 31 دستورالعمل است، که فیلد کد عمل(opcode) می باشند.) بخش پایین جدول خروجیها را به دست می دهد.

"جدول درستی" برای نوشتن ثباتی

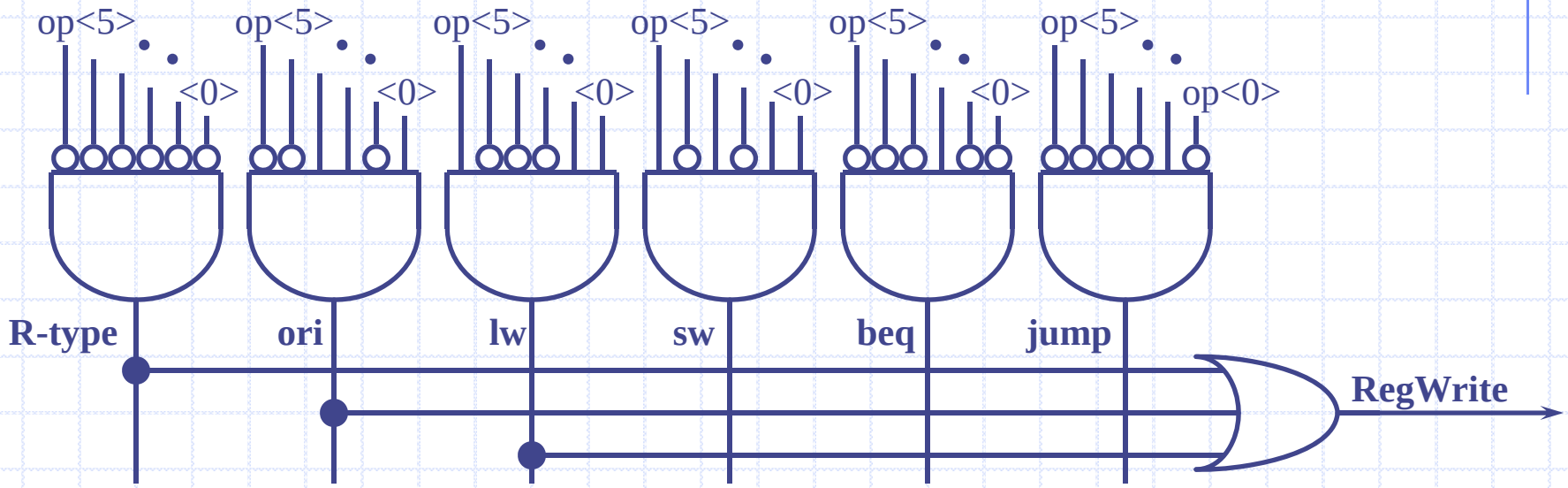
op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	**ori**	**lw**	**sw**	**beq**	**jump**
RegWrite	1	1	1	0	0	0

RegWrite = R-type + ori + lw

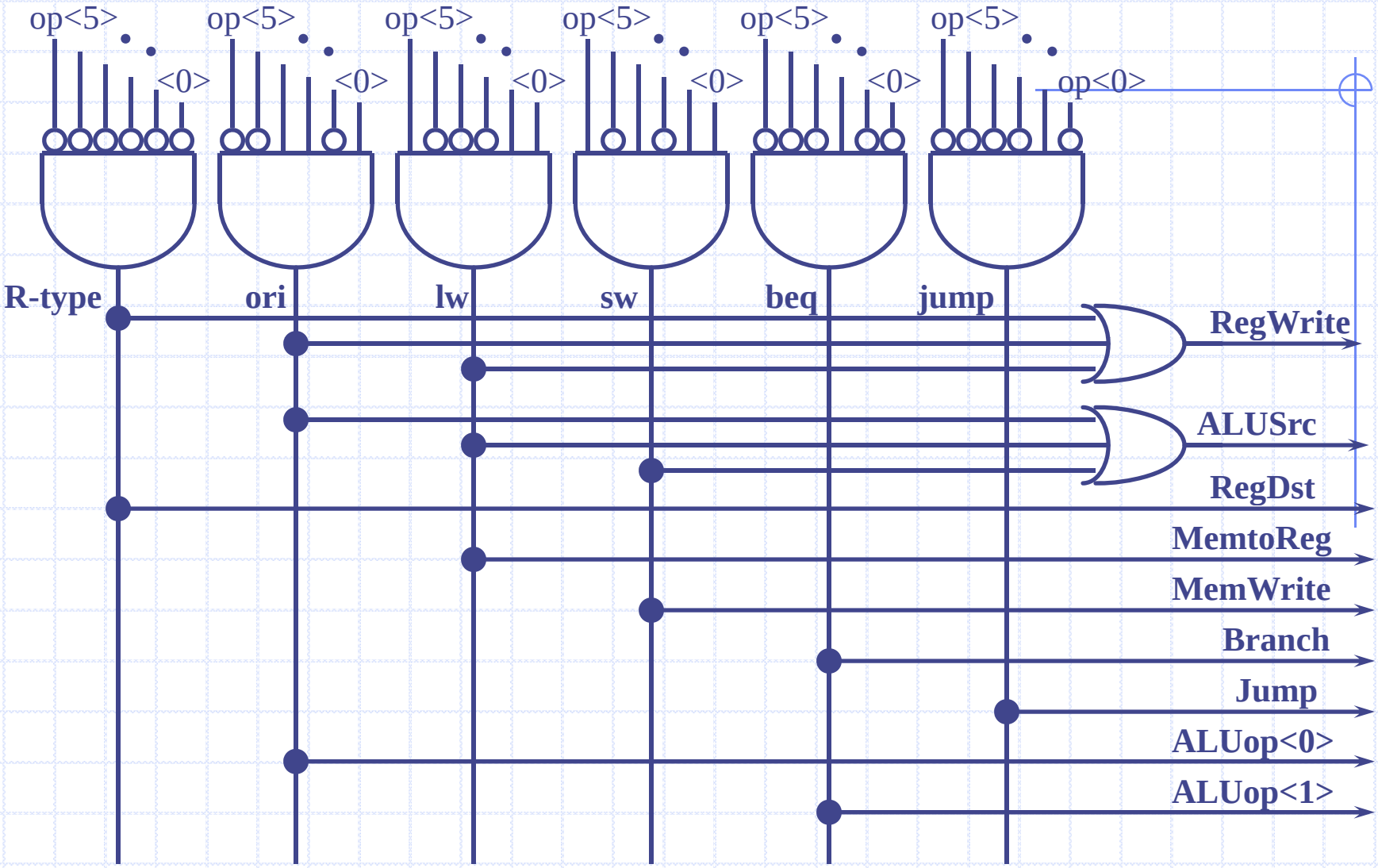
= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)

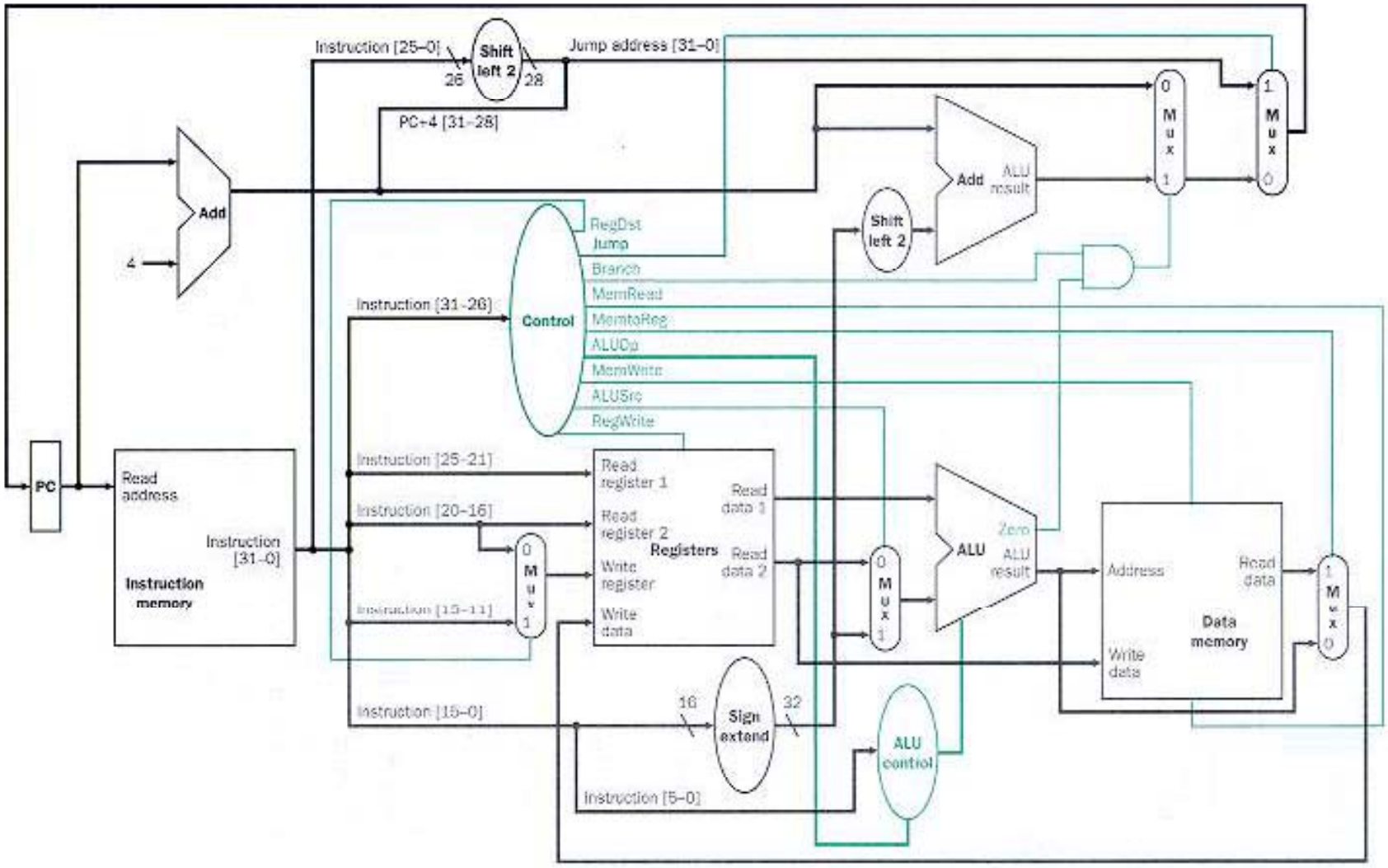
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



روشهای کنترل اصلی PLA





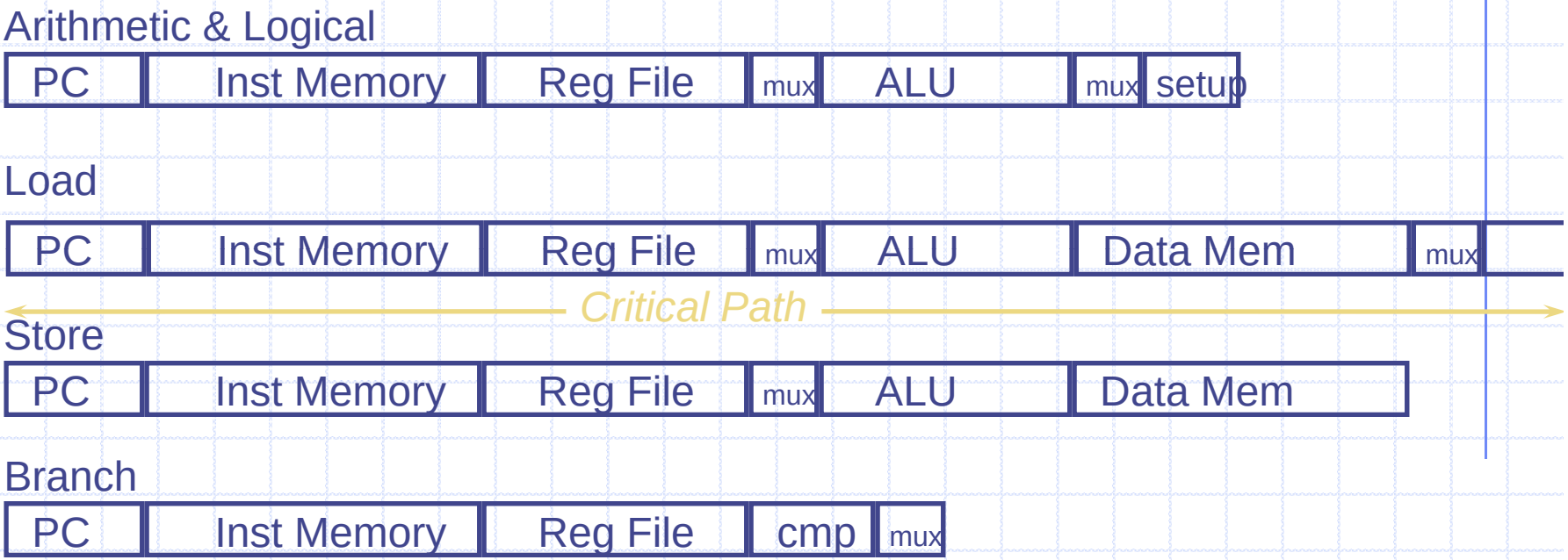
کنترل و مسیر داده ساده که برای اجرای دستورات پرش گسترش یافته است

پیاده سازی یک دستورالعمل چند چرخه ای

توضیحاتی از پیاده سازی تک چرخه ای

- دستورالعملهای ماشین ممکن است دارای طولهای متفاوت مسیر بحرانی باشند
  - دستورالعمل Load
  - دستورات ممیز شناور
  - روشهای آدرس دهی متفاوت
- زمان چرخه ساعت بوسیله بدترین مسیر بحرانی معین می شود.
- تکثیر FU ممکن است هزینه دار باشد.

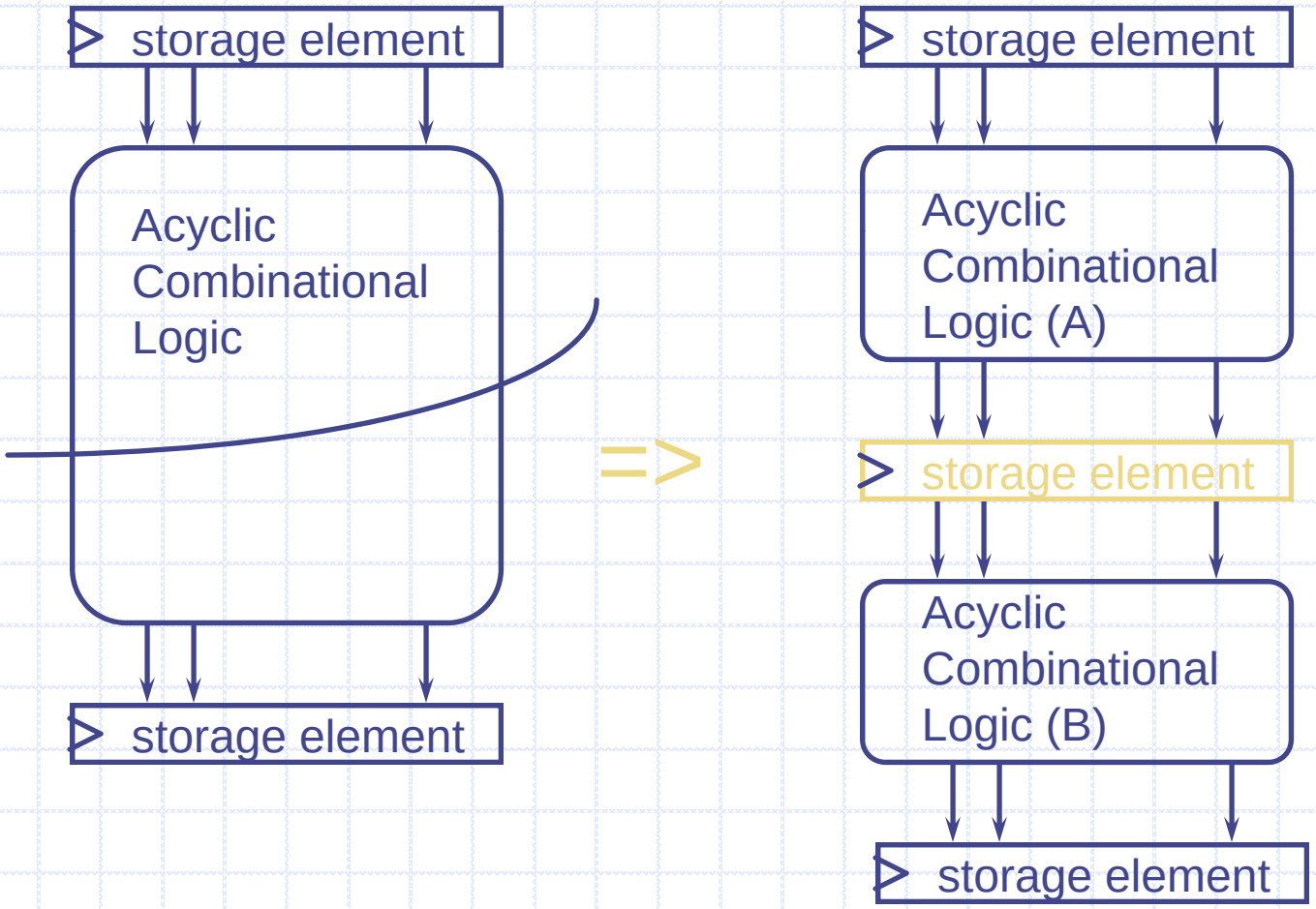
چه چیزی باعث مشکل می شود زمانی که CPI=1 پردازشگر است



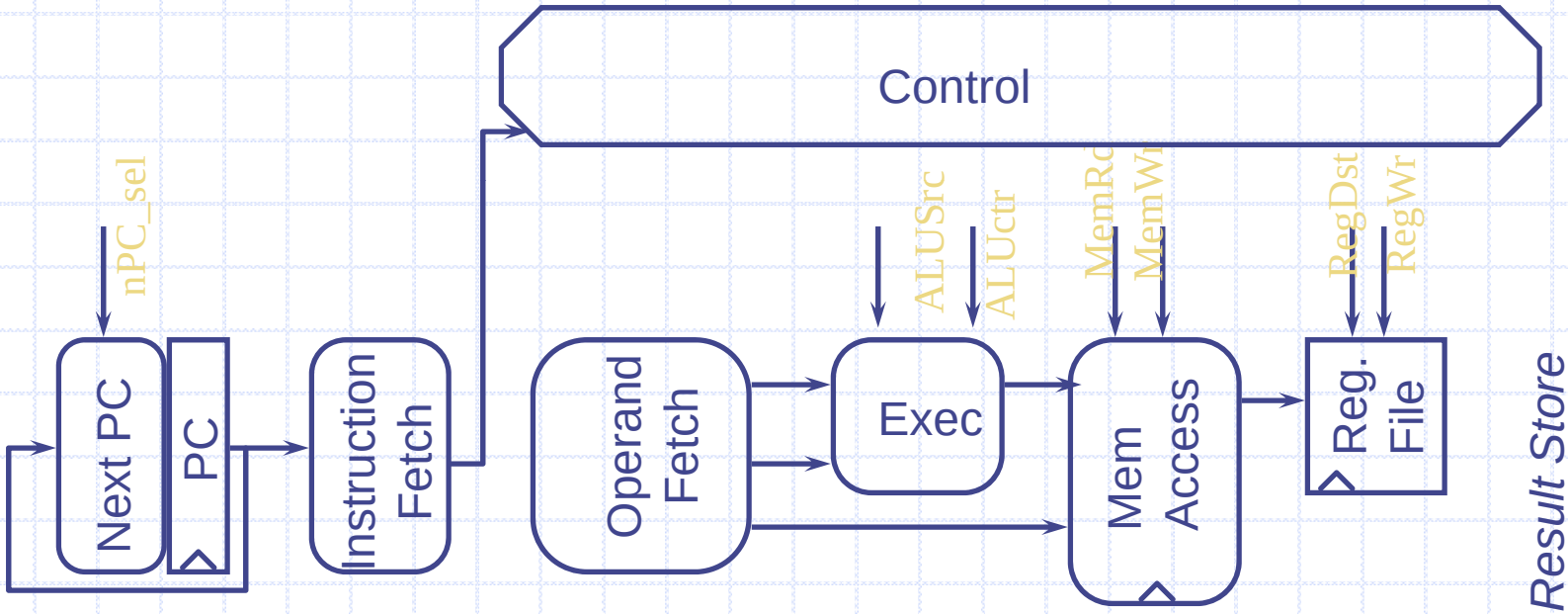
- طولانی بودن زمان چرخه
- همه دستورالعمل ها به اندازه کندترین دستور زمان می برند
- Real memory is not so nice as our idealized memory
  - cannot always get the job done in one (short) cycle

کاهش زمان چرخه

- Cut combinational dependency graph and insert register / latch
- درست همان کار رادر 2 چرخه سریع نسبت به آن که 1 چرخه کند بود انجام میدهد



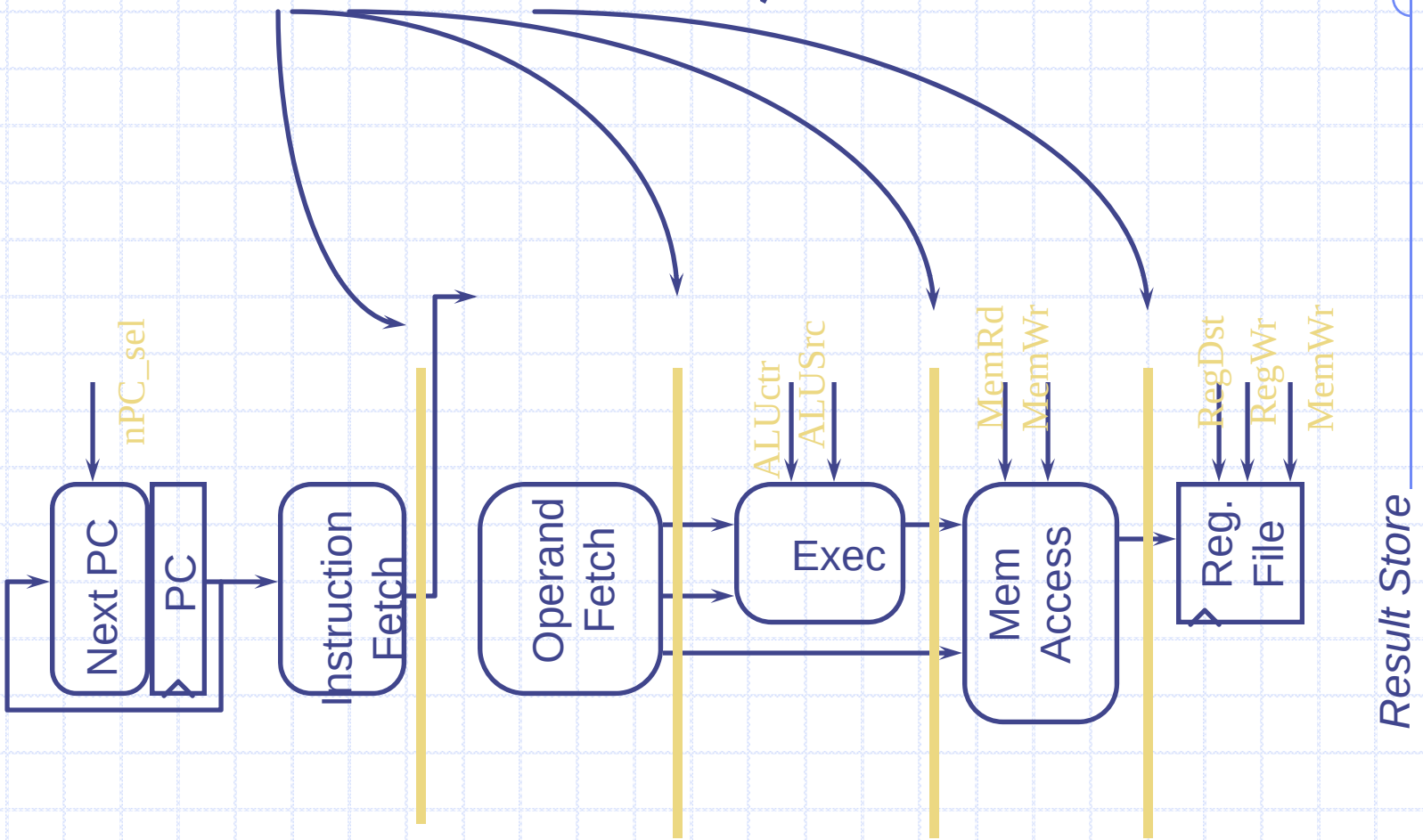
محدودیت های بنیادی زمان چرخه



تقسیم بندی 1=CPI

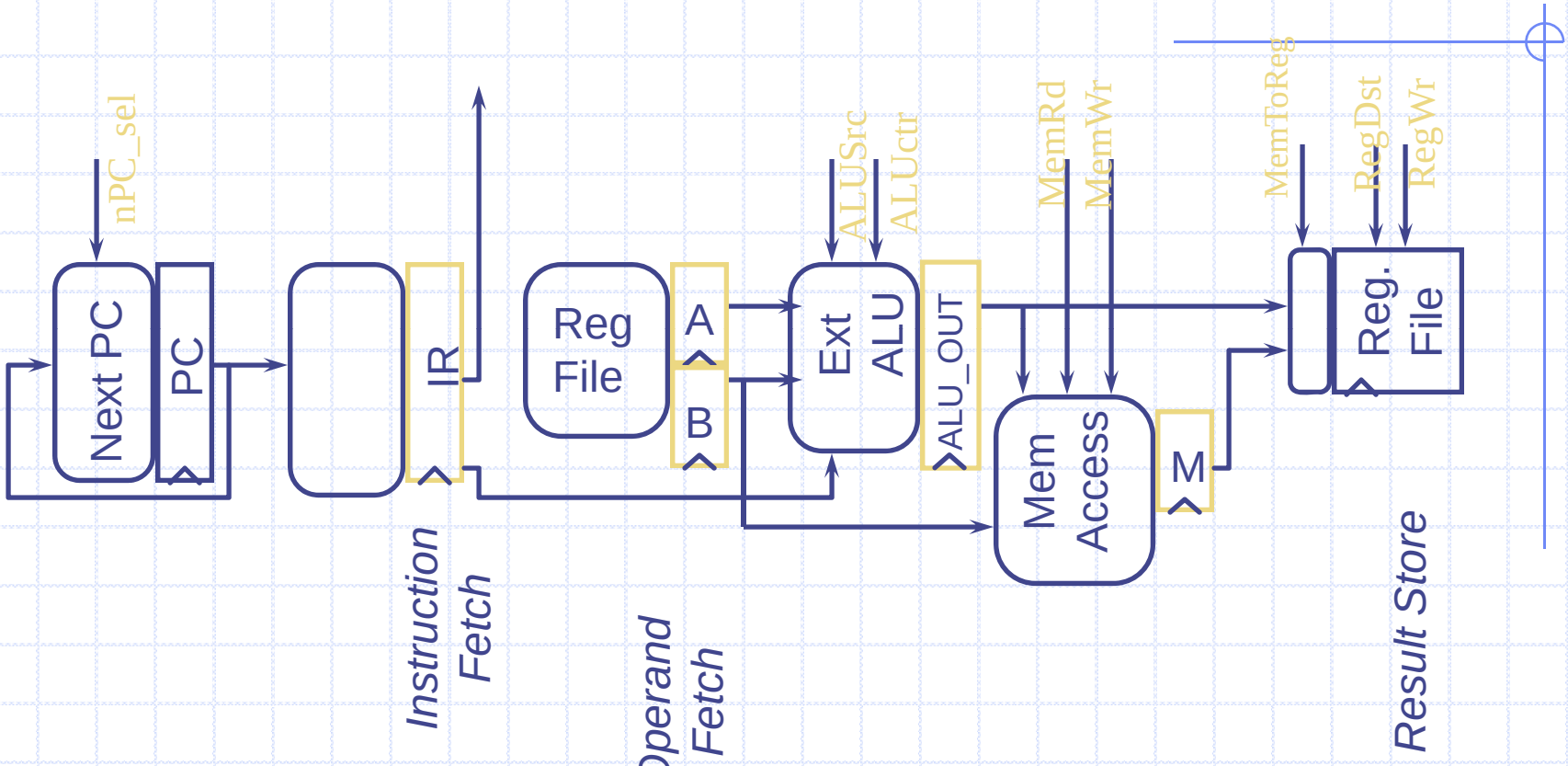
- اضافه کردن ثباتها بین کوچکترین مرحله ها

مسیر داده



امکان انجام دستورالعمل در چند چرخه

مثال مسیر داده چند چرخه ای



- ثباتهای اضافی اضافه میشوند برای ذخیره کردن مقادیر بین مراحل

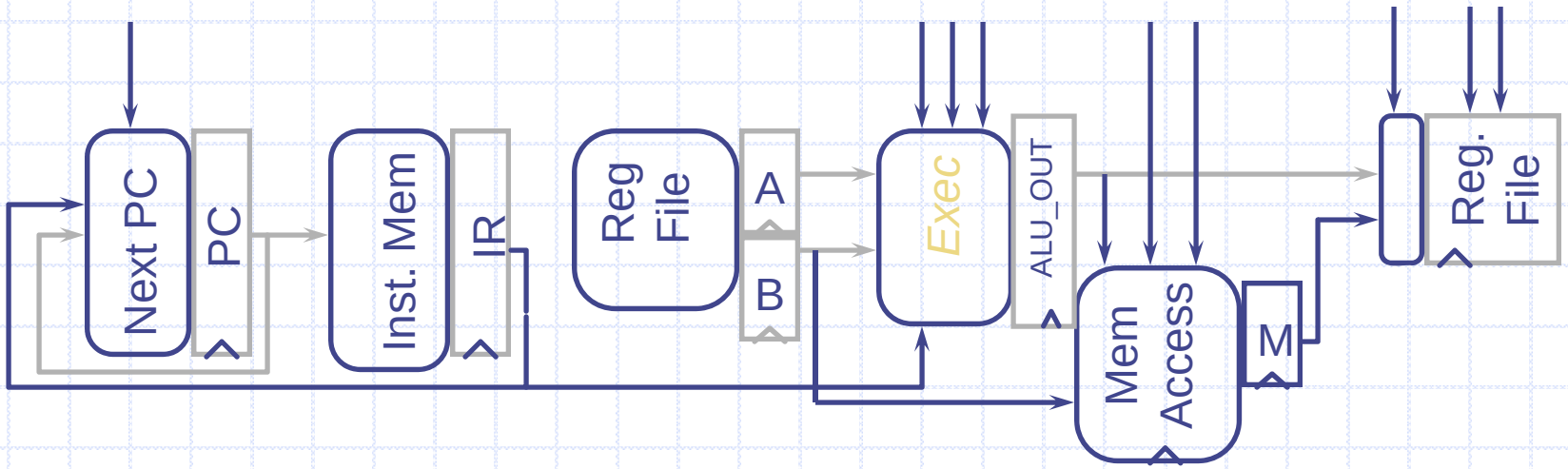
خصوصیات

- یک واحد حافظه یکسان برای دستورات و داده ها استفاده می شود.
- یک ثبات دستورالعمل به کار می رود.(IR)
- یک ALU واحد

R-type دستورالعمل های(add, sub, . . .)

inst	Logical Register Transfers
ADDU	R[rd] <– R[rs] + R[rt]; PC <– PC + 4

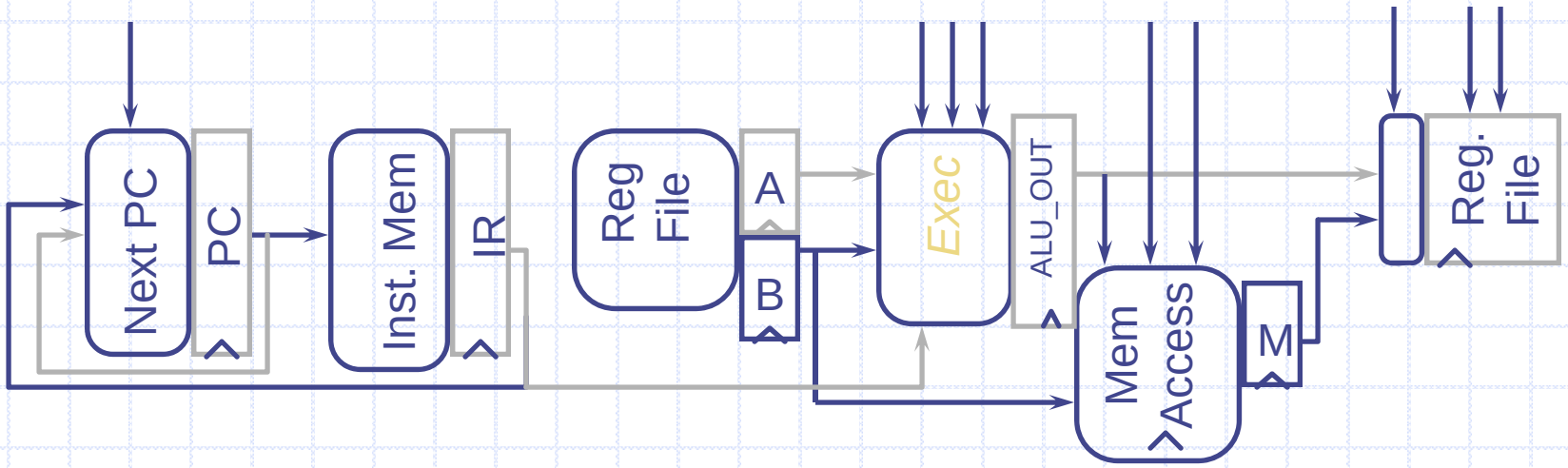
inst	Physical Register Transfers	
	IR <– MEM[pc]	
ADDU	A<– R[rs]; B <– R[rt]	
	S <– A + B	
	R[rd] <– S;	PC <– PC + 4



دستورالعمل های فوری منطقی

inst	Logical Register Transfers
ADDU	R[rt] <– R[rs] OR sx(Im16); PC <– PC + 4

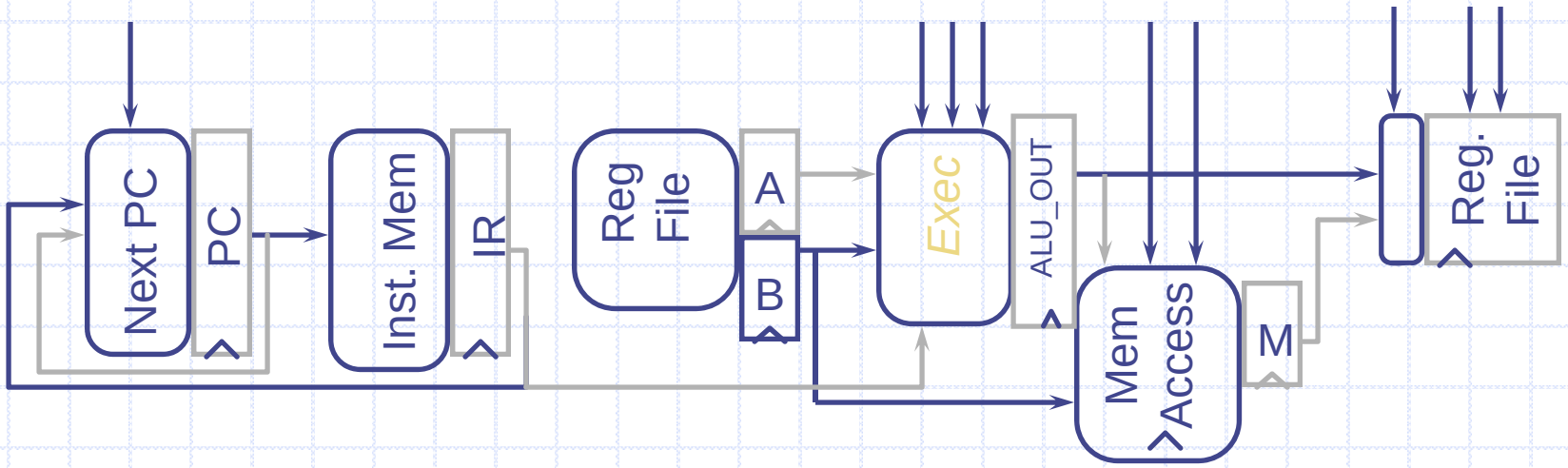
inst	Physical Register Transfers	
	IR <– MEM[pc]	
ADDU	A<– R[rs]; B <– R[rt]	
	S <– A *or* SignExt(Im16)	
	R[rt] <– S;	PC <– PC + 4



دستورالعمل بارکردن

inst	انتقال دادن منطقی ثبات
LW	R[rt] <– MEM(R[rs] + sx(Im16);
	PC <– PC + 4

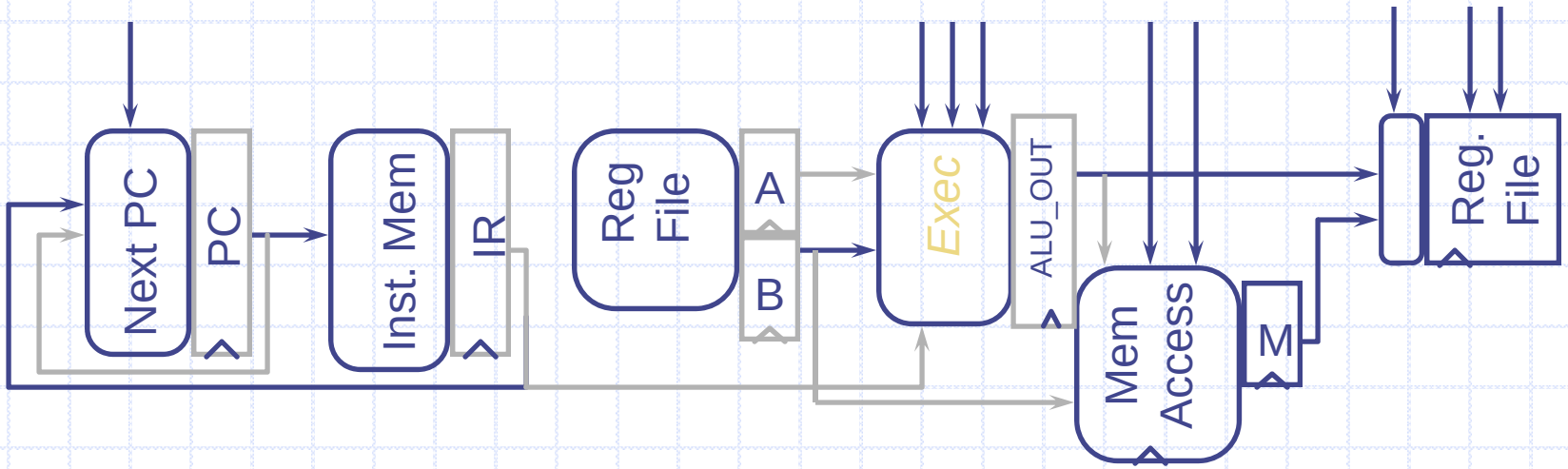
inst	انتقال دادن فیزیکی ثبات	
	IR <– MEM[pc]	
LW	A<– R[rs]; B <– R[rt]	
	S <– A + SignEx(Im16)	
	M <– MEM[S]	
	R[rd] <– M;	PC <– PC + 4



دستور العمل ذخیره کردن

inst	انتقال منطقی ثبات
SW	MEM(R[rs] + sx(Im16) <– R[rt];
	PC <– PC + 4

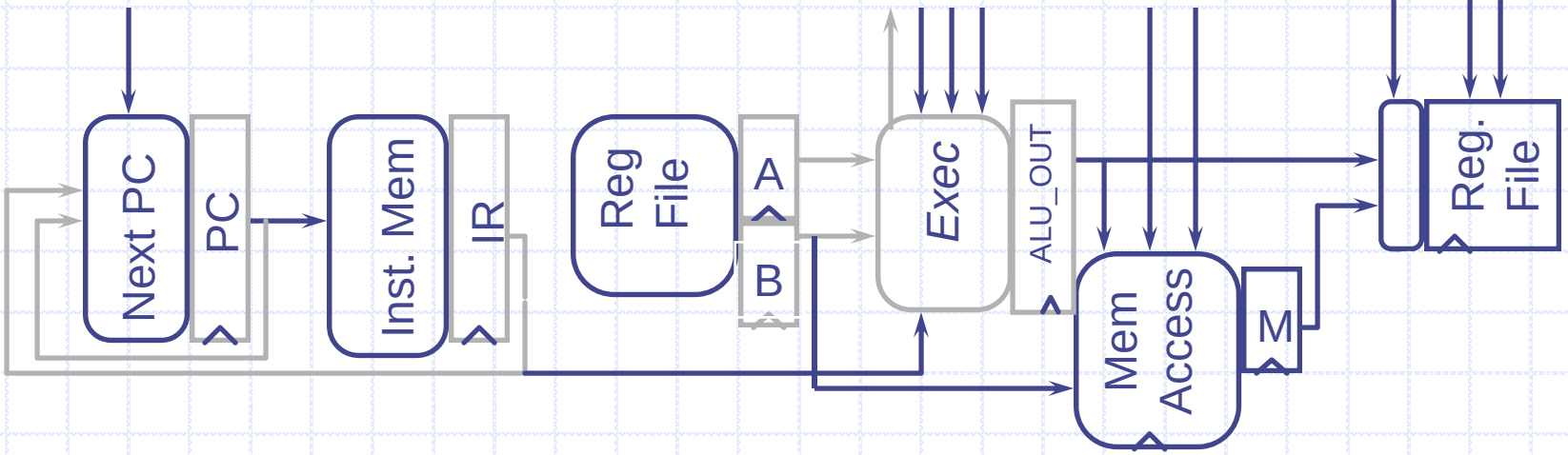
inst	انتقال فیزیکی ثبات	
	IR <– MEM[pc]	
SW	A<– R[rs]; B <– R[rt]	
	S <– A + SignEx(Im16);	
	MEM[S] <– B	PC <– PC + 4



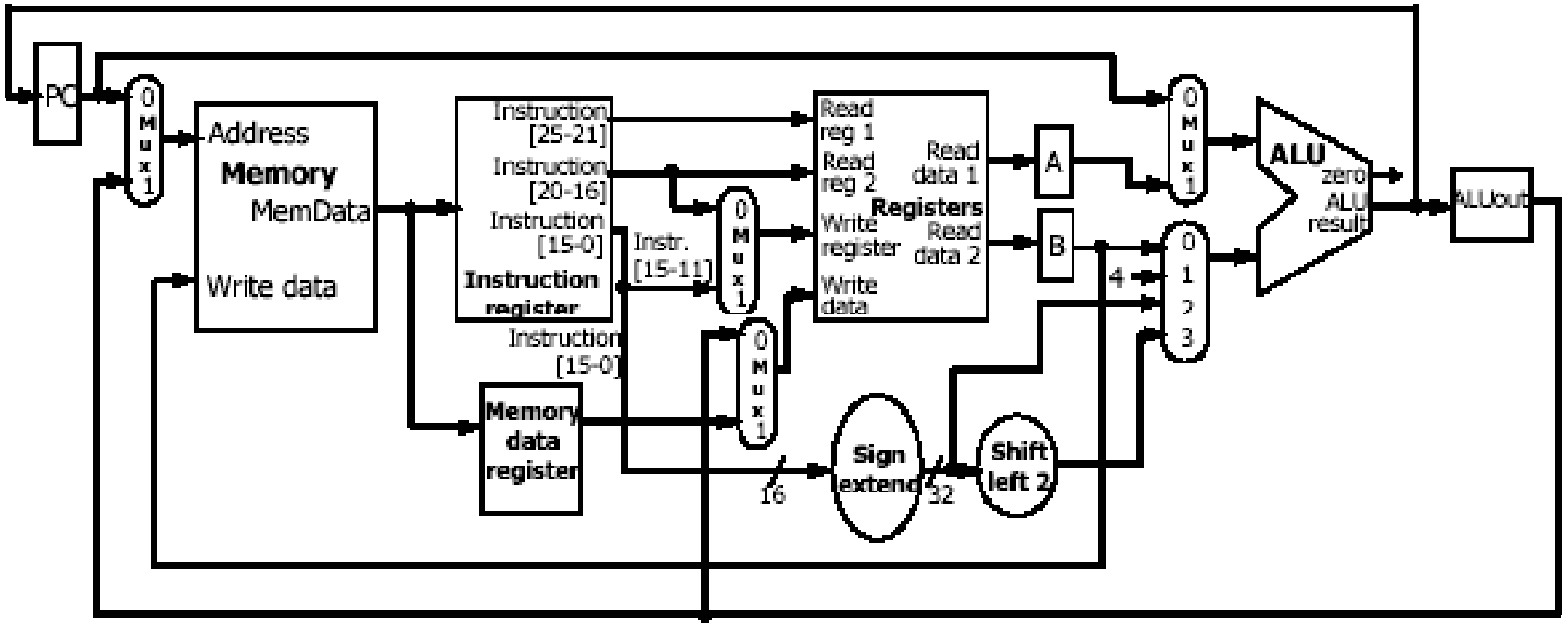
دستور العمل انشعاب

inst	انتقال منطقی ثبات
BEQ	if R[rs] == R[rt]
	then PC <= PC + sx(Im16) || 00
	else PC <= PC + 4

inst	انتقال فیزیکی ثبات
	IR <– MEM[pc]
	A<– R[rs]; B <– R[rt]
	Eq =(A - B == 0)
BEQ&Eq	PC <– PC + sx(Im16) || 00



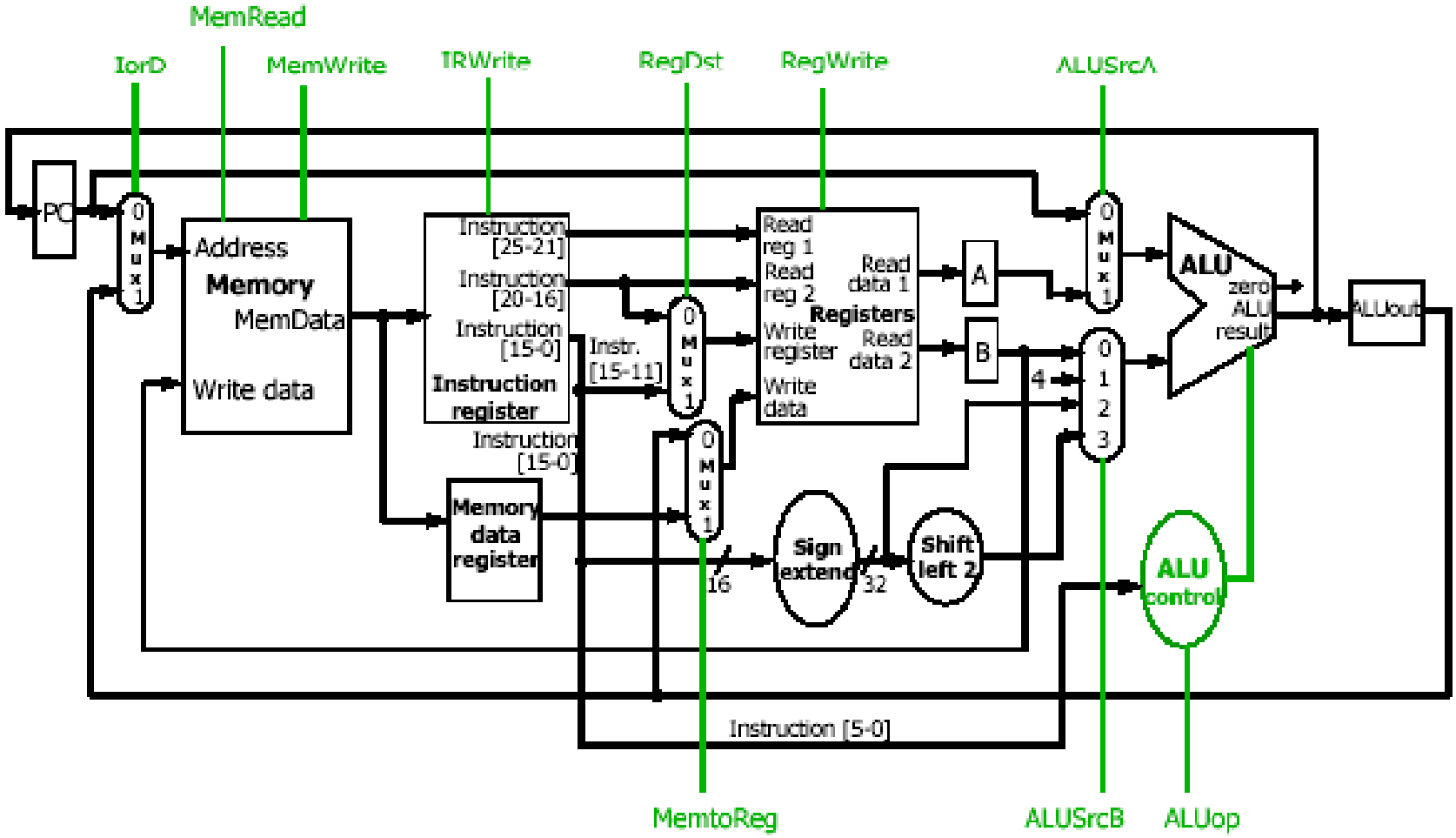
مسیر داده چند چرخه ای (Figure 5.26, p.327)



تفاوت های بین مسیر داده تکچرخه ای و چند چرخه ای

یک واحد حافظه تنها برای هر دو دستورالعمل و داده ها استفاده می شود

مسیر داده چند چرخه ای به همراه نمایش خطوط کنترلی



Signal name	Effect when deasserted	Effect when asserted
MemRead	None	Contents of memory at the read address are put on read data output.
MemWrite	None	Memory contents at the write address is replaced by value on write data input.
ALUSelA	The first ALU operand is the PC.	The first ALU operand comes from the register given by the rs field.
RegDst	The register destination number for the register write comes from the rt field.	The register destination number for the register write comes from the rd field.
RegWrite	None	The register given by Write register number is written with the value on the write data input.
MemtoReg	The value fed to the register write data input comes from the ALU.	The value fed to the register write data input comes from the Data memory.
IorD	The PC is used to supply the address to the memory unit.	The output of the ALU is used to supply the address to the memory unit.
IRWrite	None	The value from the memory unit is written into the instruction register (IR).

عمل سیگنالهای کنترلی 1 بیتی مشخص می باشد.

اعمال سیگنالهای کنترلی

Signal name	Value	Effect
ALUSelB	00	The second input to the ALU comes from the register given by the rt field.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended and shifted lower 16 bits of the IR.
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs an subtract operation.
	10	The function code field of the instruction determine the ALU operation.

عمل سیگنالهای کنترلی 2 بیتی مشخص می باشد.

اعمال سیگنالهای کنترلی

طرح ساعت زنی

- ما نیاز به یک ثبات موقتی داریم زمانیکه:

1. یک سیگنال در یک چرخه محاسبه و دریک چرخه دیگر استفاده می شود و
2. ورودیهای بلاکهای کاری (functional block) که این سیگنال را تولید می کنند می توانند قبل از اینکه این سیگنال در عناصر حالت ذخیره شود تغییر کنند.

- مثال:
  - ما به IR نیاز داریم
  - خروجی ALU
  - ثباتهای A و B به ورودیهای ALU

مراحل پایه محاسبه دستورالعمل

1. IFمرحله: واکشی دستورالعمل
IR= Memory [PC]
PC= PC + 4

2. ID step کد برداری و واکشی عملوندها
A= ثبات;[IR(25-21)]
B= ثبات;[IR(20-16)]
Target= PC + (علامت گسترش یافته;[IR(15-0)<< 2]

یادداشت:ما می خواهیم عملیات را"خوشبینانه"انجام دهیم. those are common to all, or at least do no hurt any.

(cont'd)

مراحل پایه محاسبه دستورالعمل

EX .3 محاسبه . محاسبه آدرس حافظه ، یا تکمیل انشعاب

نمونه:

ارجاع به حافظه:

ALU= A + sign-extend [IR(15-0)]; خروجی

دستورالعمل های ALU :

= A op B;خروجیALU

انشعاب:

If (A ==B) PC= Target;

(cont'd)

مراحل پایه محاسبه دستورالعمل

4. MEM دسترسی به حافظه یا تکمیل دستورالعمل R__TYPE

نمونه:

ارجاع به حافظه:

MDR = Memory [ALUoutput];

Memory[ALUoutput] = B; or

دستورالعمل های ALU:

Reg[IR(5-11)] = ALUoutput;

5. WB : بازنوشتن

Reg[IR(20-16)] =MDR

کنترل برای PCبعدی

PCبعدی تعیین می شود به وسیله:

- زمانی که PCبرای واکشی دستورالعمل ترتیبی یک واحد افزوده شده خروجیALUیک منبع است.
- ثبات هدف ، منبع است زمانی که دستورالعمل داده شده یک انشعاب شرطی باشد ما همچنین به یک سیگنال برای نوشتن در ثبات نیاز خواهیم داشت که به آنTarget Writeگفته می شود.

PCSource:یک سیگنال کنترلی 2 بیتی برای بالا

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches
Instruction fetch	IR = Memory[PC] PC = PC + 4		
Instruction decode/ register fetch	A = Registers[IR(25-21)] B = Register[IR(20-16)] Target = PC + (sign-extend [IR(15-0)] <<2)		
Execution, address computation, or branch completion	ALUoutput = A op B	ALUoutput = A + sign-extend [IR(15-0)]	if (A == B) then PC = Target
Memory access or R-type completion	Reg[IR(15-11)] = ALUoutput	memory-data = Memory[ALUoutput] or Memory [ALUoutput] = B	
Write-back		Reg[IR(20-16)] = memory-data	

خلاصه ای از مراحلی که برای اجرای هر نوع دستورالعمل انجام می شود.

تعداد مراحل اجرای دستورالعمل ها 3 تا 5 مرحله است .دو مرحله اول مستقل از نوع دستورالعمل هستند .بعد از این مراحل ، یک دستورالعمل از 1 تا حداکثر 3 چرخه برای تکمیل شدن زمان می برد که این وابسته به نوع دستورالعمل است

Signal name	Effect when deasserted	Effect when asserted
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written; if the Zero output from the ALU is also active.
TatgetWrite	None	The output of the ALU is written into the register Target.

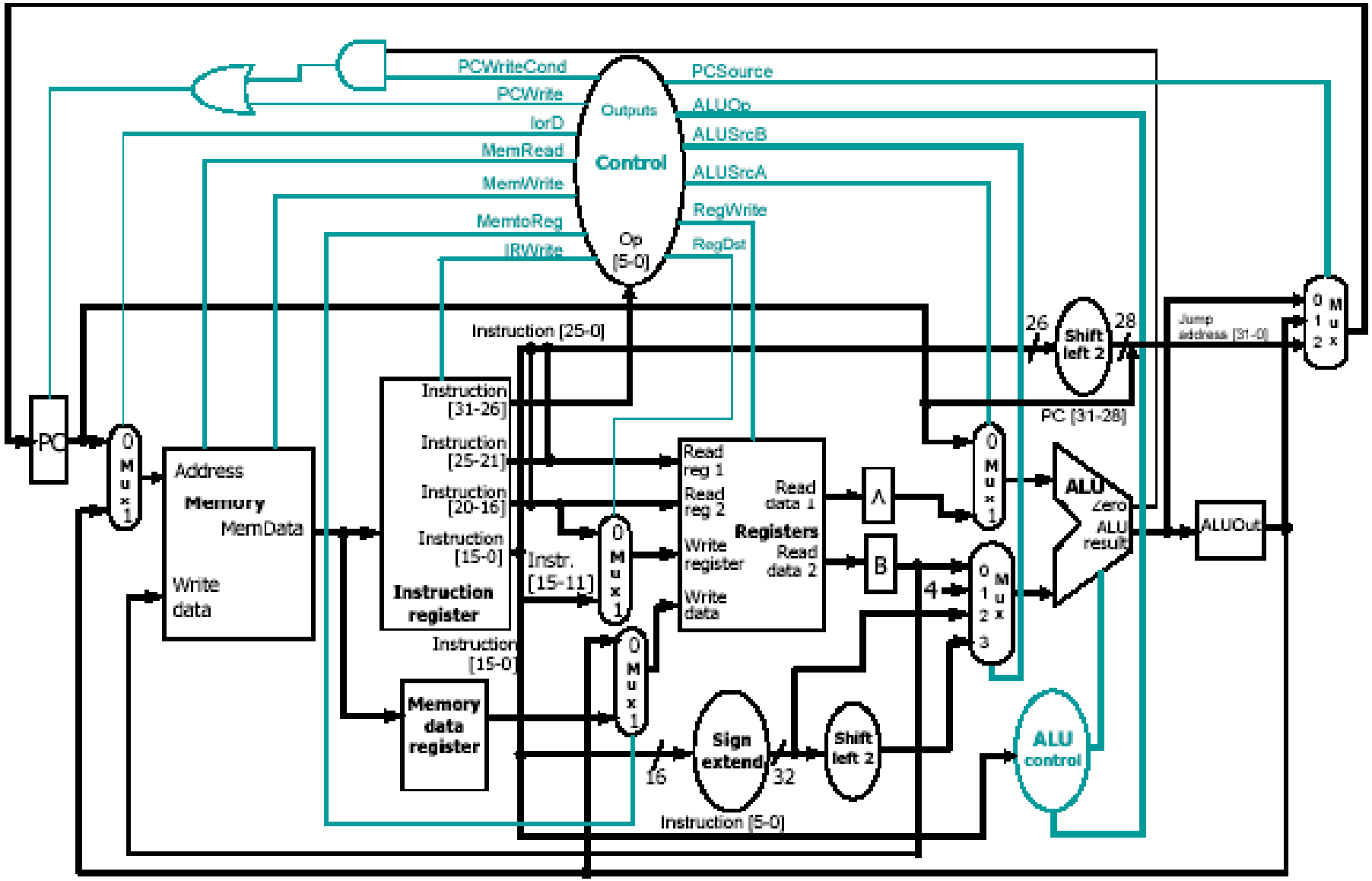
عمل ناشی از تنظیم هریک از سیگنال های کنترلی 1 بیتی

Signal name	Value	Effect
	00	The ALU output is sent to the PC for writing.
PCSource	01	The contents of the register Target are sent to the PC for writing.
	10	The jump target address [PC + 4(29-26)] concatenated with IR (25-0) and shifted left two bits) is sent to the PC for writing.

عمل ناشی از تنظیم هریک از سیگنال های کنترلی 2 بیتی

کنترل کردن pcبعدی

شکل5.28:مسیر داده و سیگنال های کنترلی کامل برای چند چرخه ای(همراه پرش)

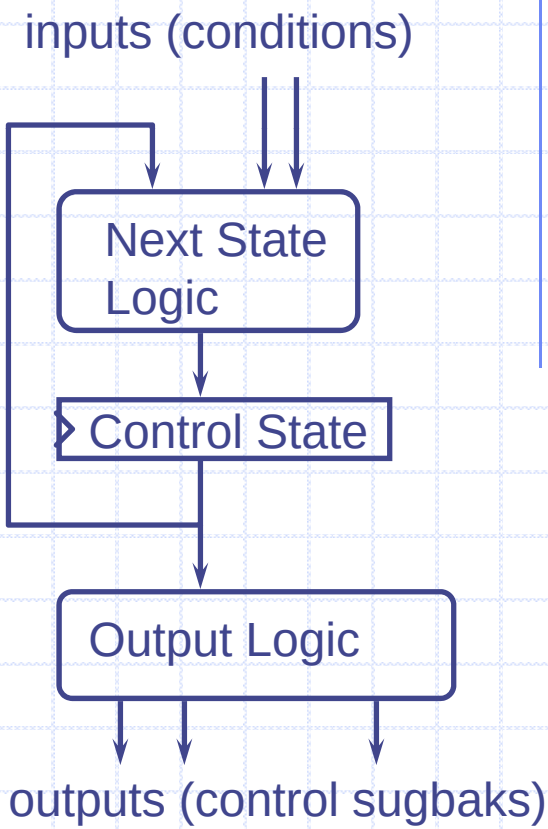


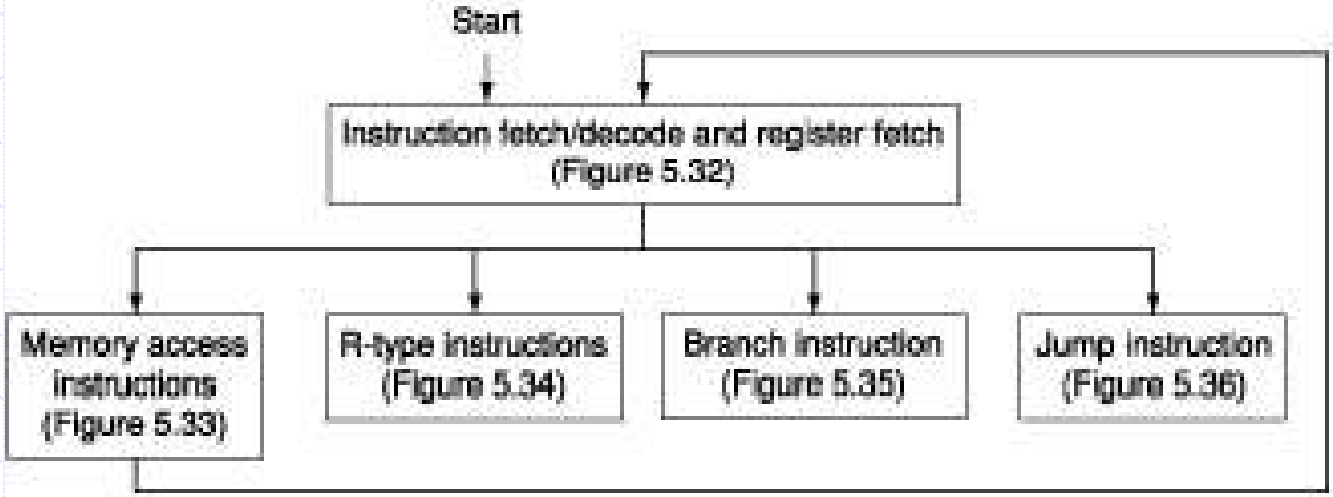
روشهای طراحی کنترل برای اجرای دستورات چند چرخه ای

- مبنا قرار دادن ماشین با حالات محدود
- مبنا قرار دادن Microprogramming

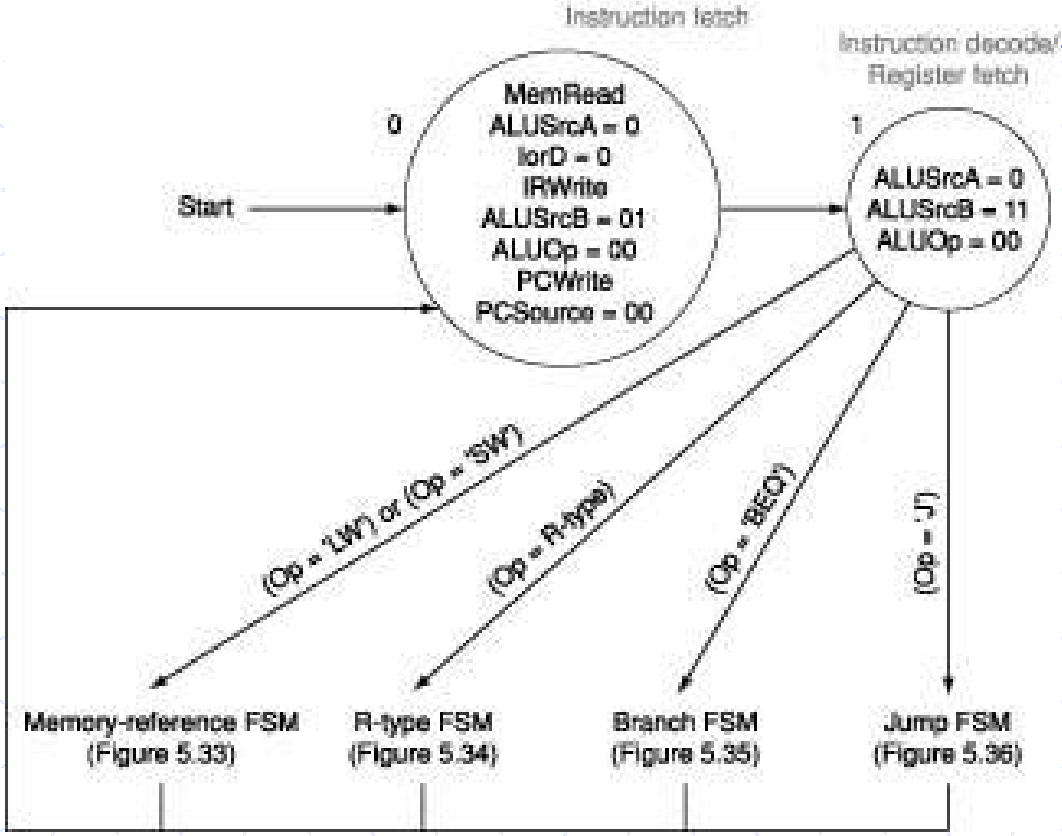
ساختارFSM

- مراحل:
  - Specify a set of output to be asserted
  - Assume signals not asserted are left disserted by default
  - کنترل کردن تسهیم کننده ها همیشه صراحتا مشخص شده است
- تابع حالت بعدی

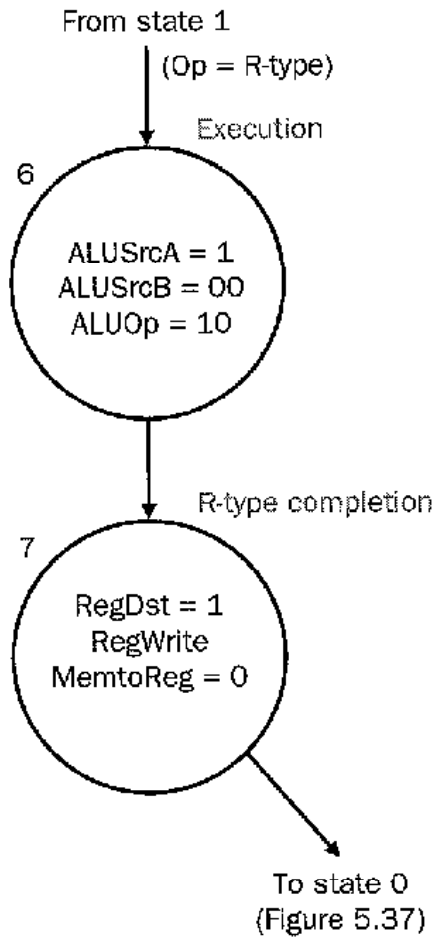




دید سطح بالایی از کنترل ماشین حالات متناهی(FSM)

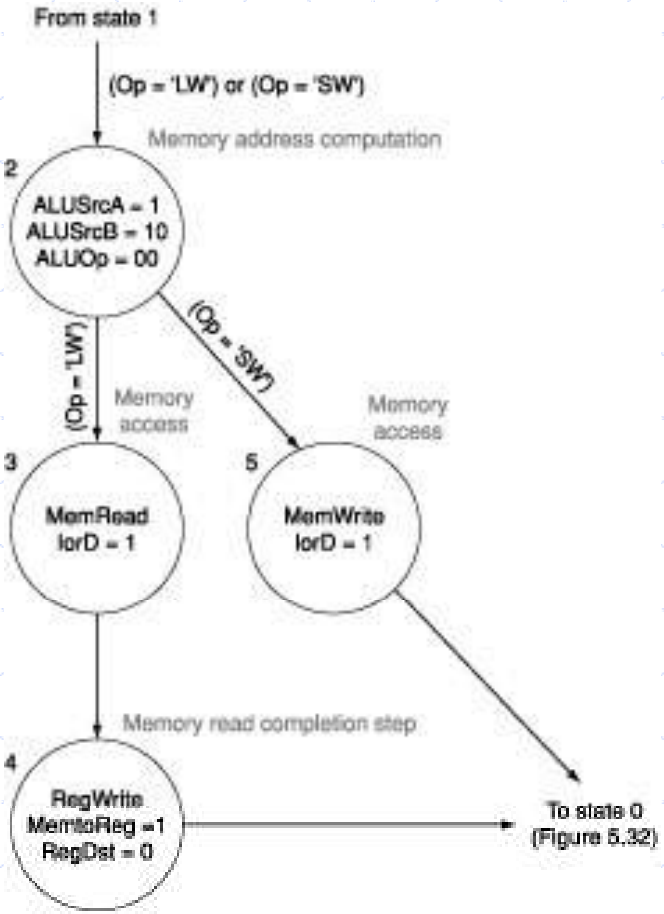


دستورالعمل واکشی و کدبرداری قسمت یکسانی برای همه دستورالعمل ها هستند(شکل 5.37_جدید 5.32)

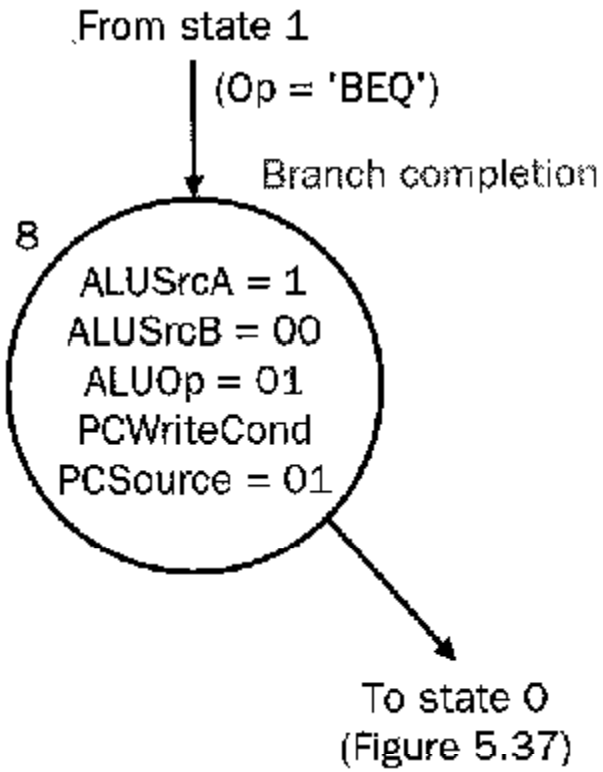


New 5.32

دستورالعمل هایR_type می توانند با یک FSM 2مرحله ای ساده پیاده سازی شوند

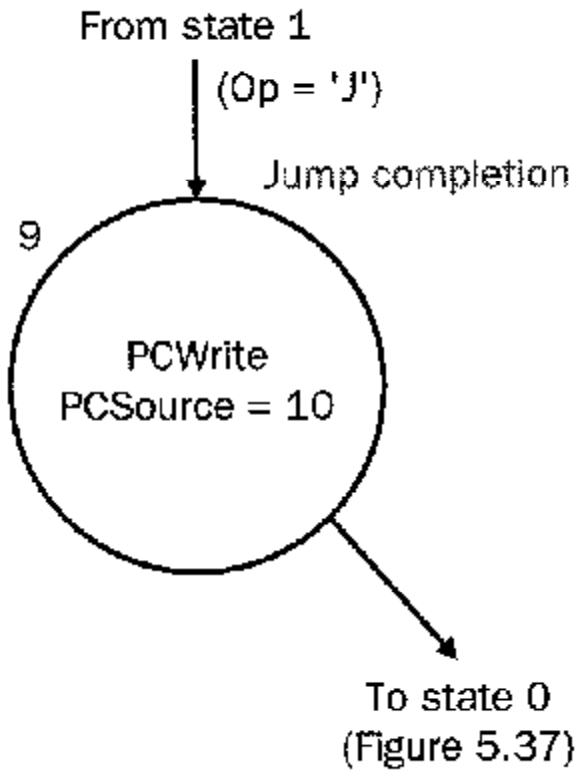


FSMبرای کنترل کردن دستورالعمل های ارجاع به حافظه 4 مرحله دارد(شکل 5.38 _جدید 5.33



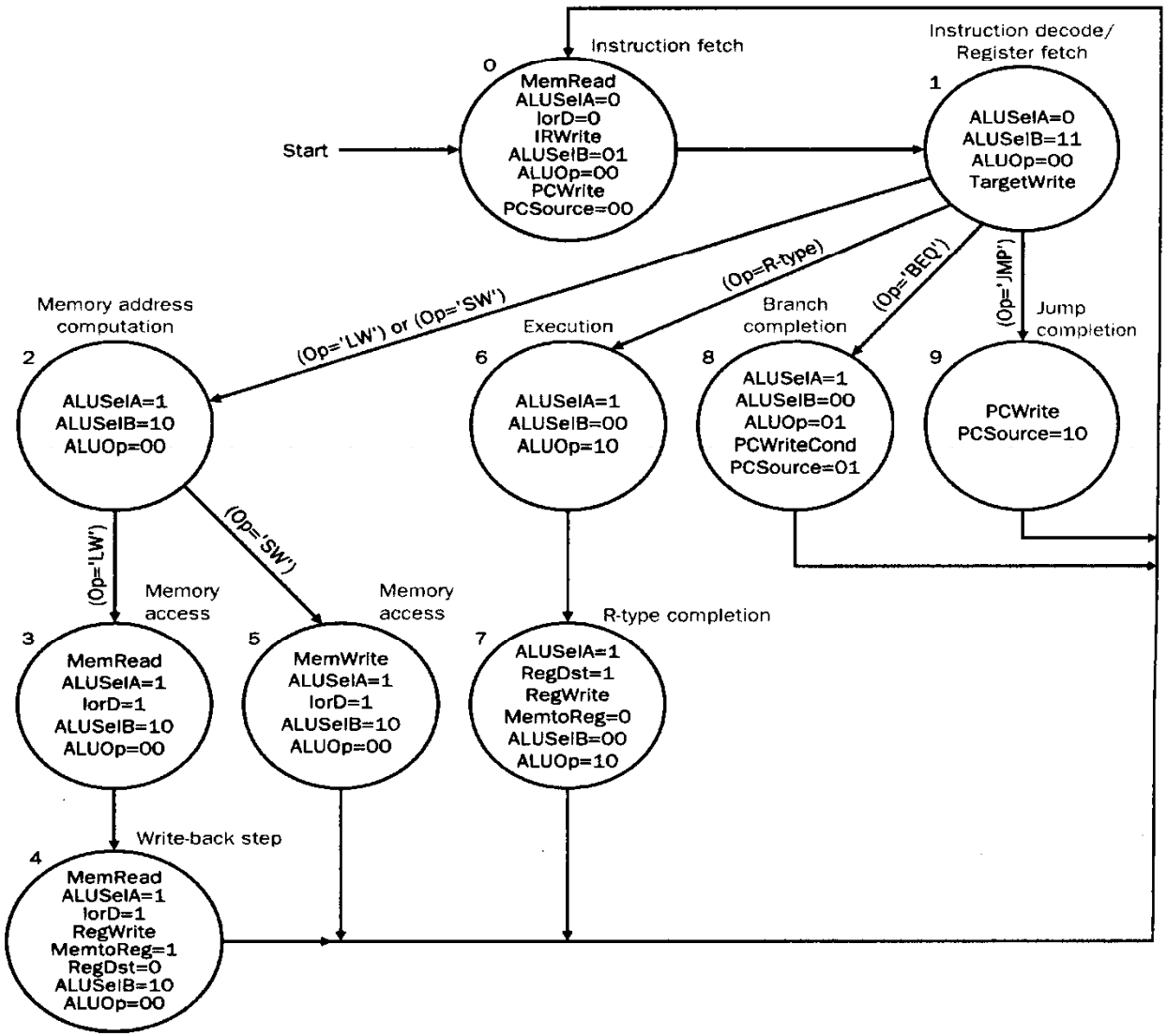
New 5.32

دستورالعمل انشعاب فقط به یک ماشین تک مرحله ای نیاز دارد(شکل5.40 جدید 5.35)



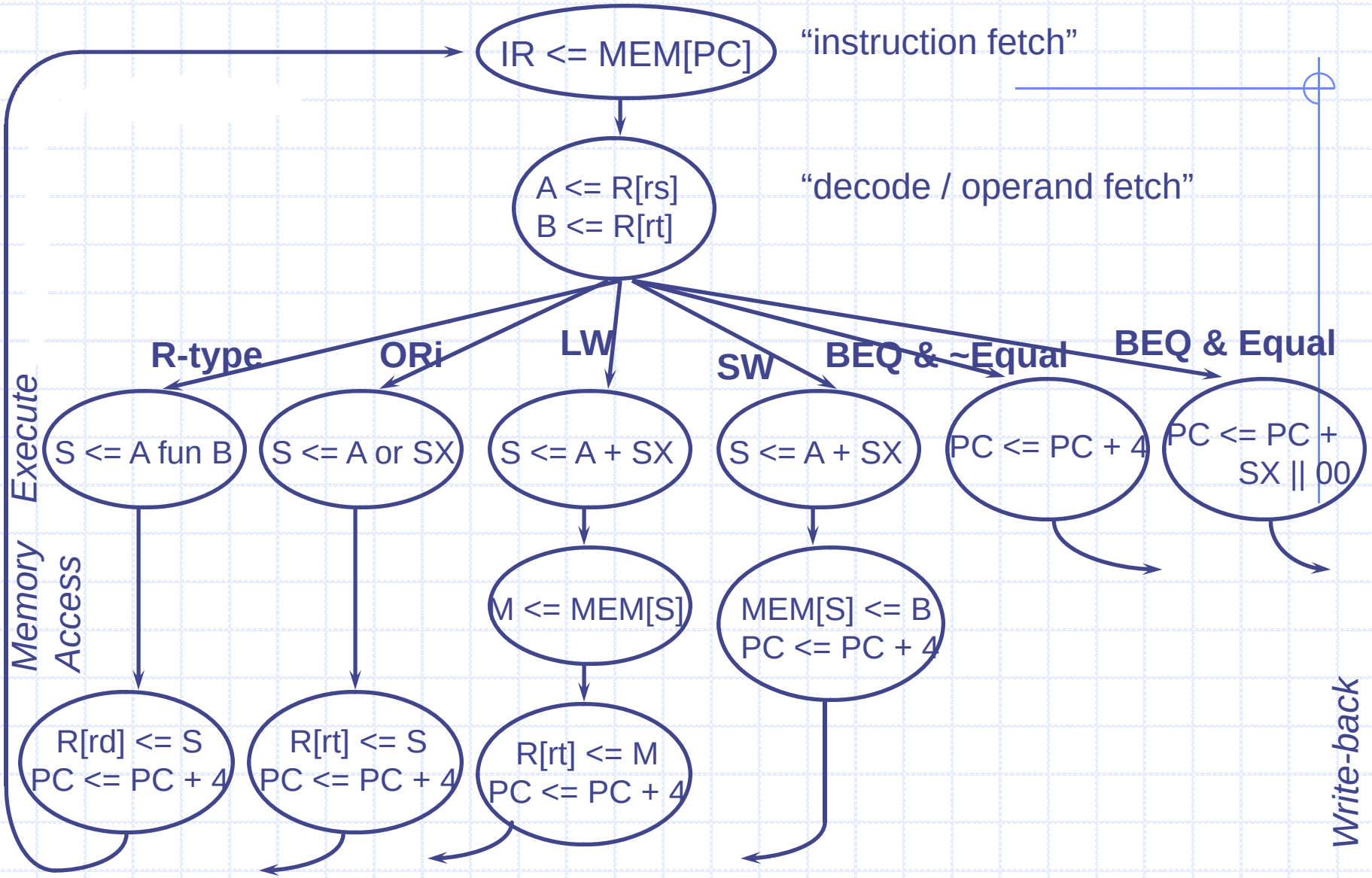
New 5.32

The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the instruction register shifted left two bits. (Figure 5.41 – new 5.36))

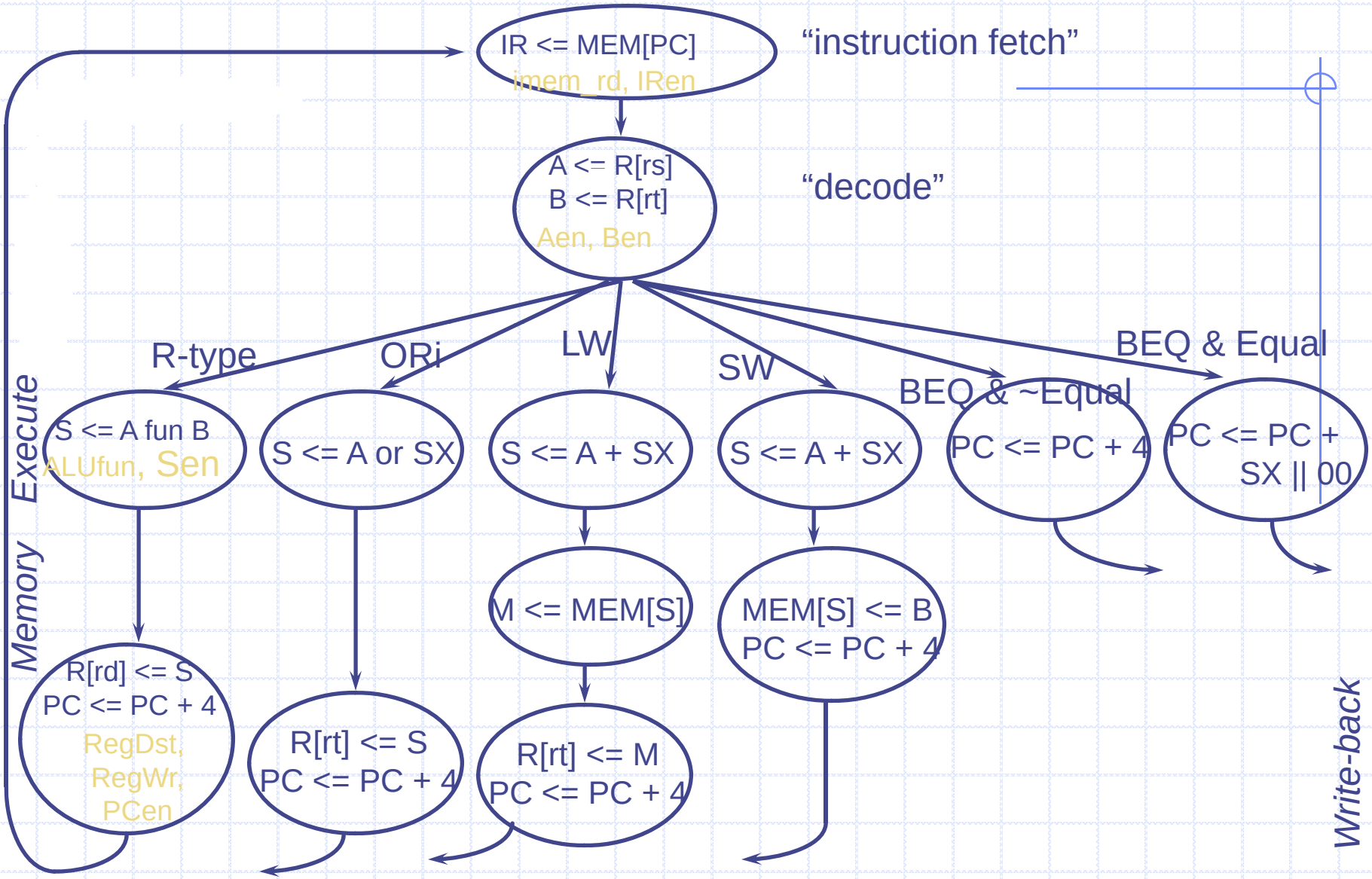


واحد کنترل ماشین حالت متناهی به صورت کامل بر روی مسیرداده ها (Figure 5.42 – New 5.38)

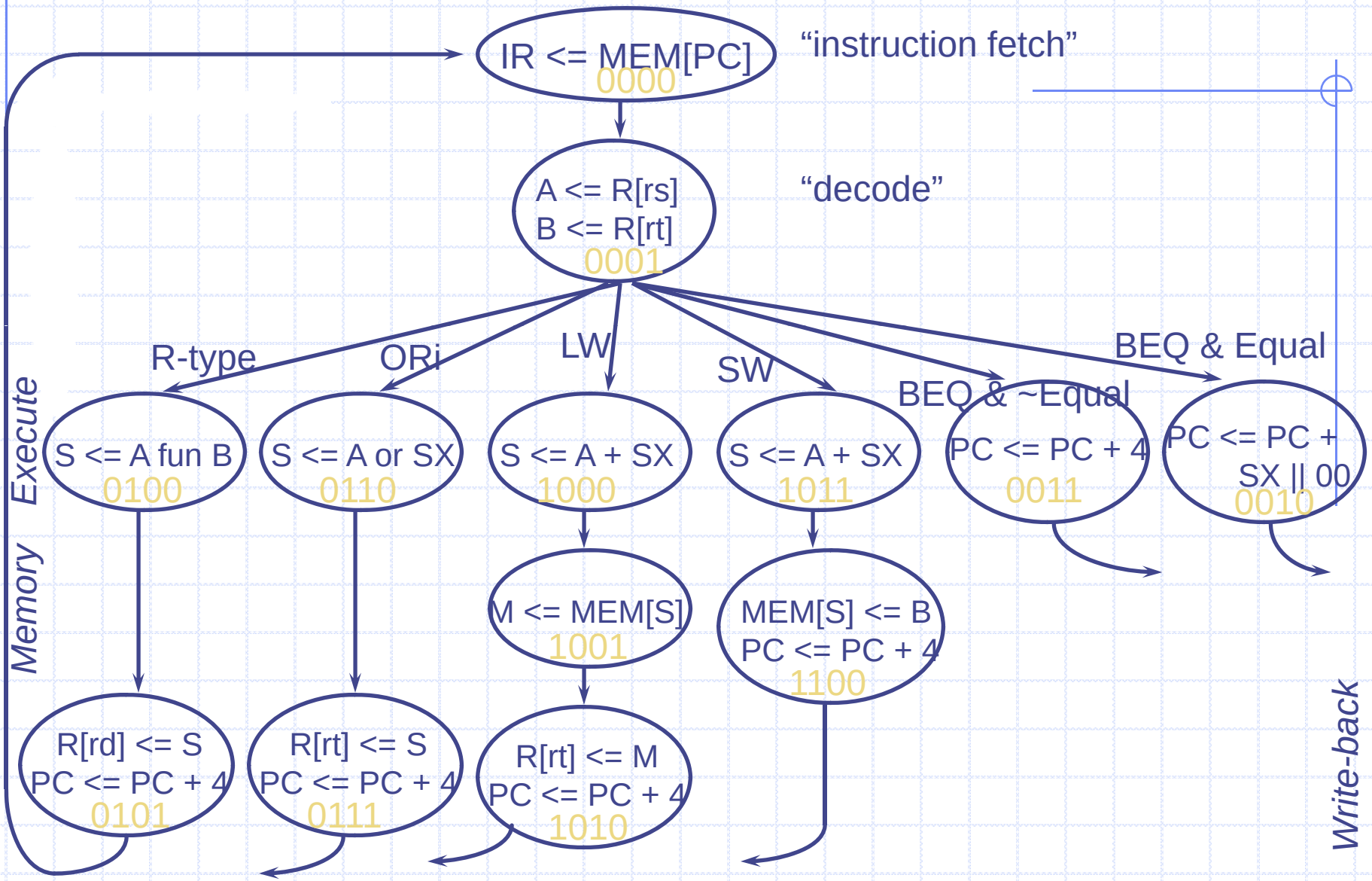
کنترل اختصاصی برای چند چرخه ای



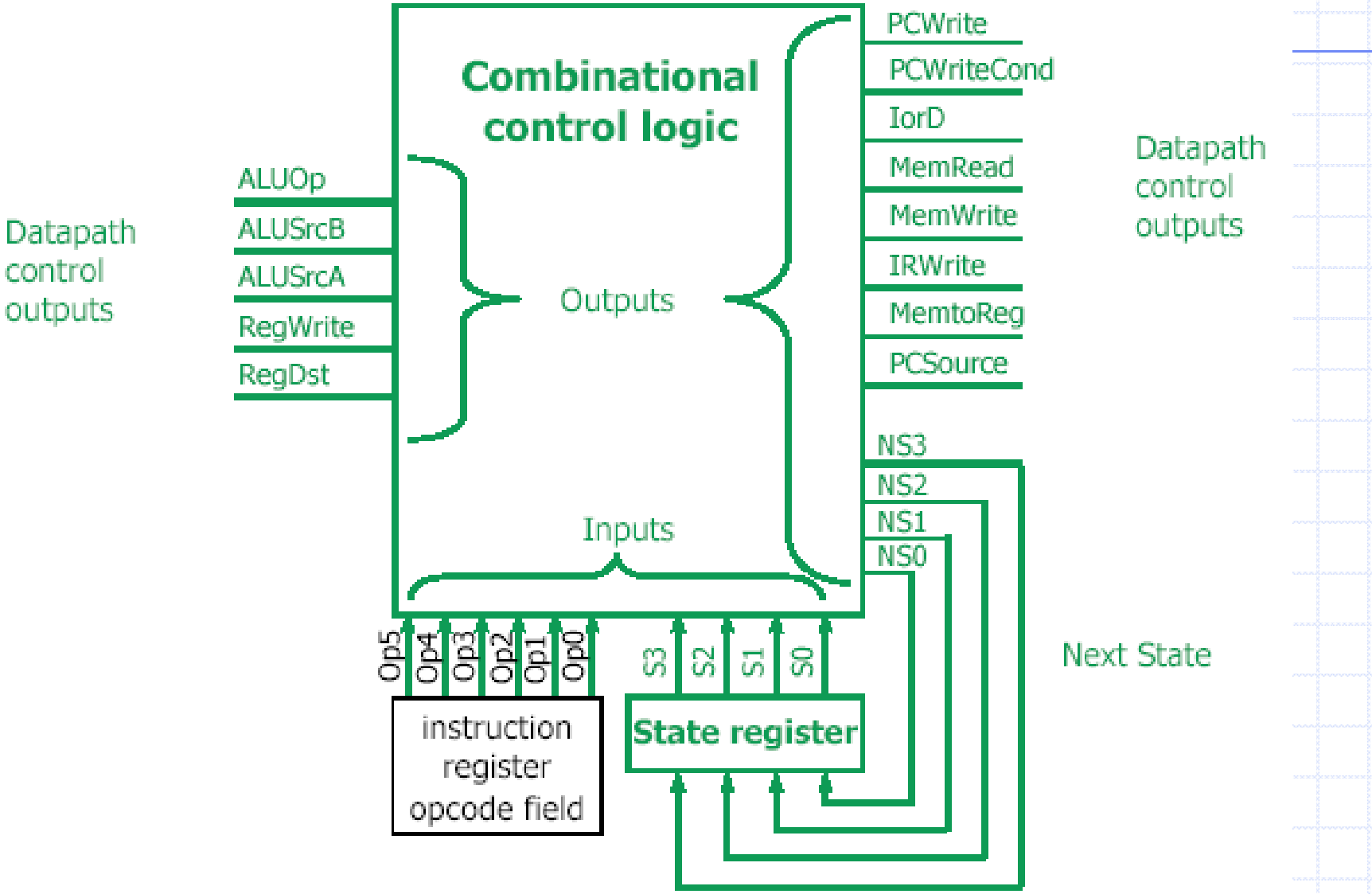
Mapping RTs to Control Points



تخصیص حالات


IR <= MEM[PC]
0000
"instruction fetch"
A <= R[rs]
B <= R[rt]
0001
"decode"
R-type
ORi
LW
SW
BEQ & Equal
BEQ & ~Equal
Execute
S <= A fun B
0100
S <= A or SX
0110
S <= A + SX
1000
S <= A + SX
1011
PC <= PC + 4
0011
PC <= PC + SX || 00
0010
Memory
M <= MEM[S]
1001
MEM[S] <= B
PC <= PC + 4
1100
R[rd] <= S
PC <= PC + 4
0101
R[rt] <= S
PC <= PC + 4
0111
R[rt] <= M
PC <= PC + 4
1010
Write-back

اجرای دستگاه کنترل ماشین حالت متناهی



تعادل منطقی برای خروجی های سیگنال کنترلی

Output	Current States
PCWrite	state0 + state9
PCWriteCond	state8
IorD	state3 + state5
MemRead	state0 + state3
MemWrite	state5
IRWrite	state0
MemtoReg	state4
PCSource1	state9
PCSource0	state8
ALUOp1	state6
ALUOp0	state8
ALUSrcB1	state1 + state2
ALUSrcB0	state0 + state1
ALUSrcA	state2 + state6 + state8
RegWrite	state4 + state7
RegDst	state7

For Example:

$$PCWrite = \overline{S3}\cdot\overline{S2}\cdot\overline{S1}\cdot\overline{S0} + S3\cdot\overline{S2}\cdot\overline{S1}\cdot S0$$

تعادل منطقی برای خروجی های حالت بعدی

Output	Current States	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jump')

For Example:

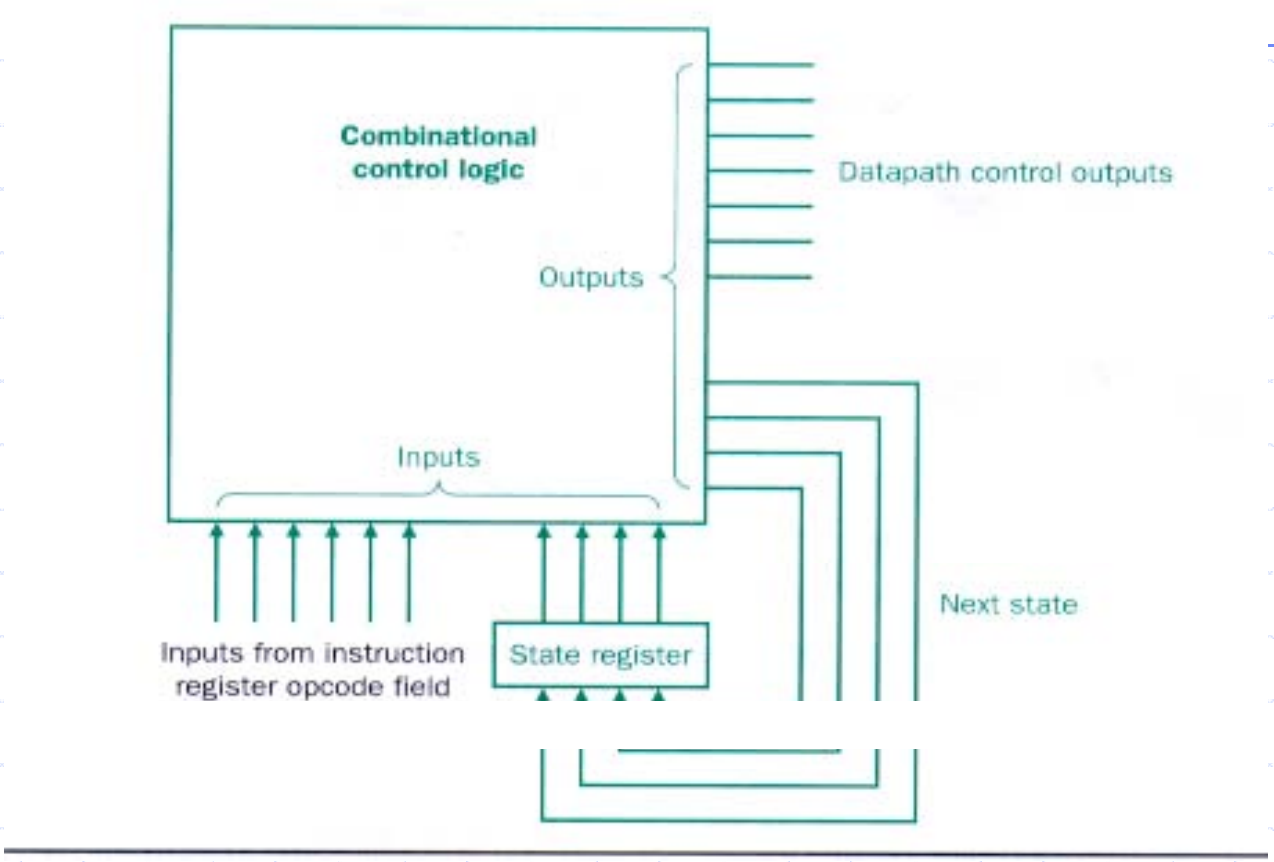
NextState1 = State0 = $\overline{S3} \cdot \overline{S2} \cdot \overline{S1} \cdot \overline{S0}$

NextState3 = State2 · (Op[5-0]='lw')

= $\overline{S3} \cdot \overline{S2} \cdot S1 \cdot \overline{S0} \cdot Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot Op2 \cdot Op1$

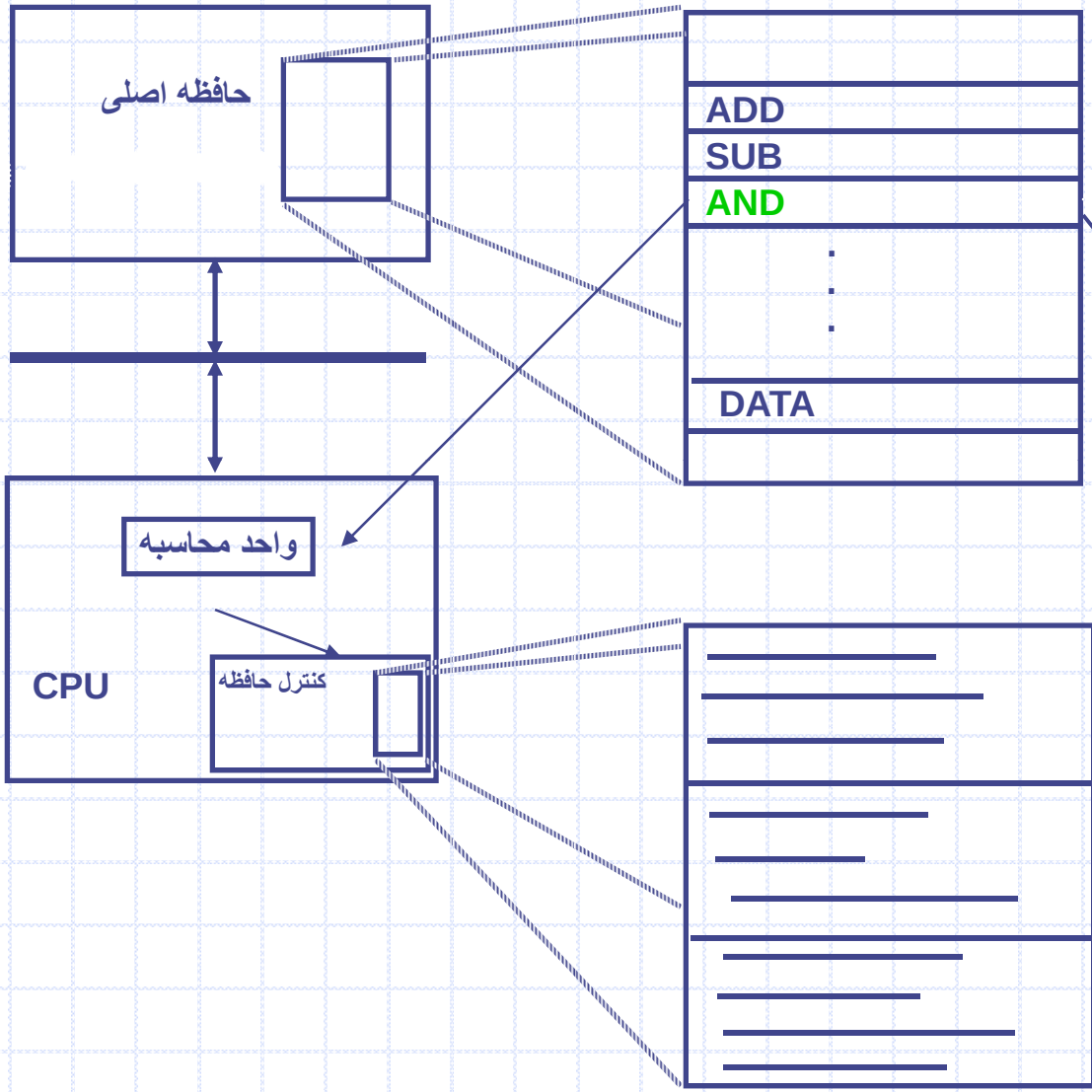
سوال:

- چه طور میشود CPIرا از روی FSAتعیین کرد؟
- FSAچه طور انجام می شود؟



Finite state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state.

تفسیر "دستور العملMacro"



برنامه کاربر به اضافه داده ها
این می تواند تغییر کند

یکی از اینها طراحی شده در یکی
از اینها

و ترتیبmicro

e.g., Fetch
Calc Operand Addr
Fetch Operand(s)
Calculate
Save Answer(s)

Field name	Values for field	Function of field with specific value
Label	Any string	Used to specify labels to control microcode sequencing. Labels that end in a 1 or 2 are used for dispatching with a jump table that is indexed based on the opcode. Other labels are used as direct targets in the microinstruction sequencing. Labels do not generate control signals directly but are used to define the contents of dispatch tables and generate control for the Sequencing field.
ALU control	Add	Cause the ALU to add.
	Subt	Cause the ALU to subtract; this implements the compare for branches.
	Func code	Use the instruction's funct field to determine ALU control.
SRC1	PC	Use the PC as the first ALU input.
	A	Register A is the first ALU input.
SRC2	B	Register B is the second ALU input.
	4	Use 4 for the second ALU input.
	Extend	Use output of the sign extension unit as the second ALU input.
	Extshft	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read	Read two registers using the rs and rt fields of the IR as the register numbers, putting the data into registers A and B.
	Write ALU	Write the register file using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	Write the register file using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	Read memory using ALUOut as address; write result into MDR.
	Write ALU	Write memory using the ALUOut as address; contents of B as the data.
PCWrite control	ALU	Write the output of the ALU into the PC.
	ALUOut-cond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	Write the PC with the jump address from the instruction.
Sequencing	Seq	Choose the next microinstruction sequentially.
	Fetch	Go to the first microinstruction to begin a new instruction.
	Dispatch i	Dispatch using the ROM specified by i (1 or 2).

واکشی برنامه میکرو

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1

دستورالعمل ارجاع به حافظه برنامه میکرو

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch

برنامه میکرو R-type

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

انشعاب برنامه میکرو

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch

پرش برنامه میکرو

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
JUMP1						Jump address	Fetch

همه میکرو کد

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

فصل ششم

خط لوله ای کردن و معماری خط لوله ای شده

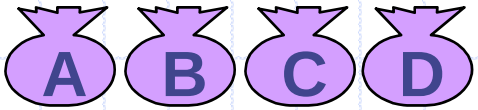
خط لوله ای

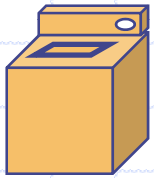
- خط لوله ای چیست – مفاهیم پایه
- مسیر داده خط لوله ای شده

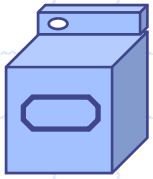
  یک بررسی موردی از MIPS
- کنترل خط لوله
- رفع hazard در خط لوله

مفاهیم پایه
خط لوله ای

خط لوله ای کردن، معمول است

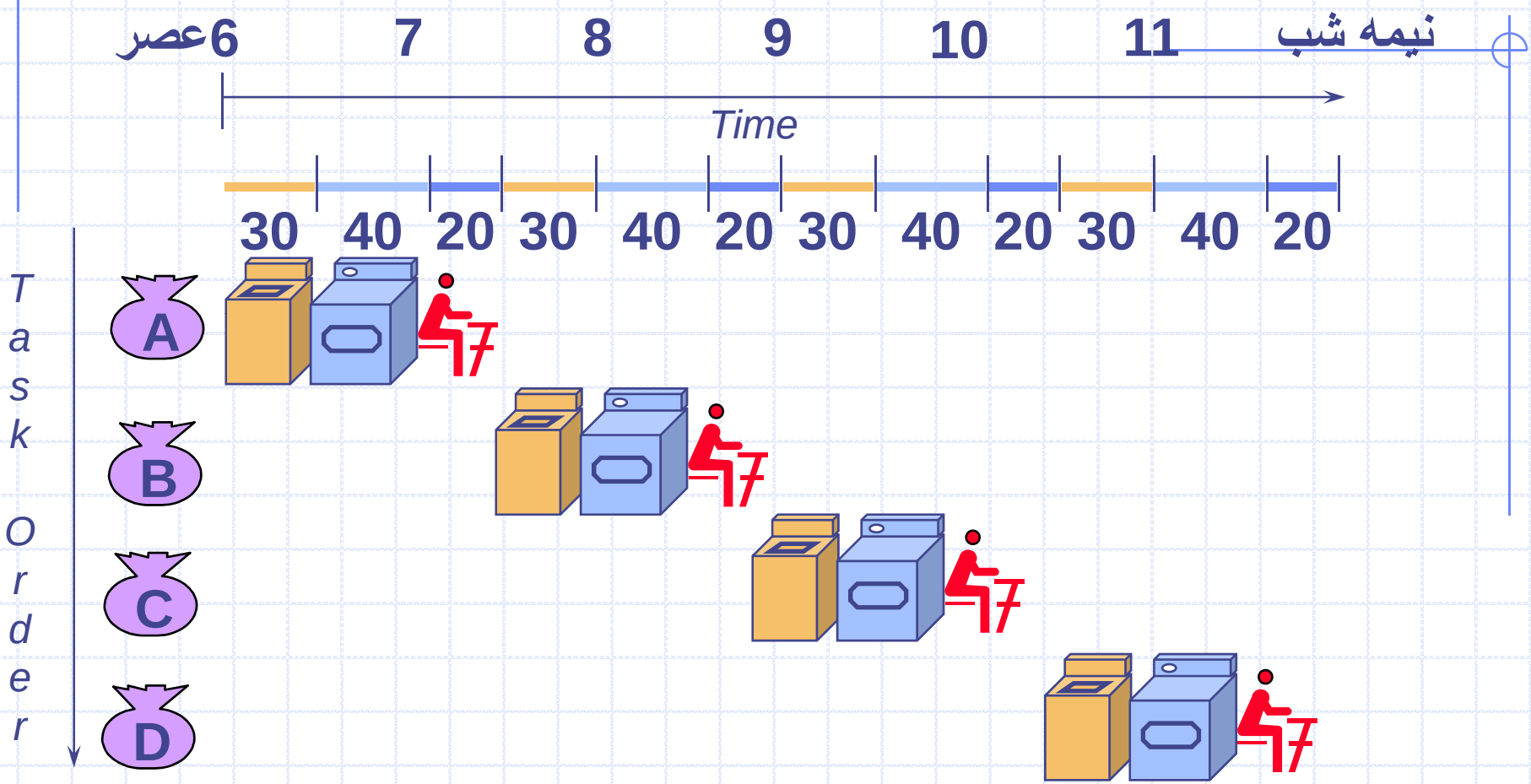






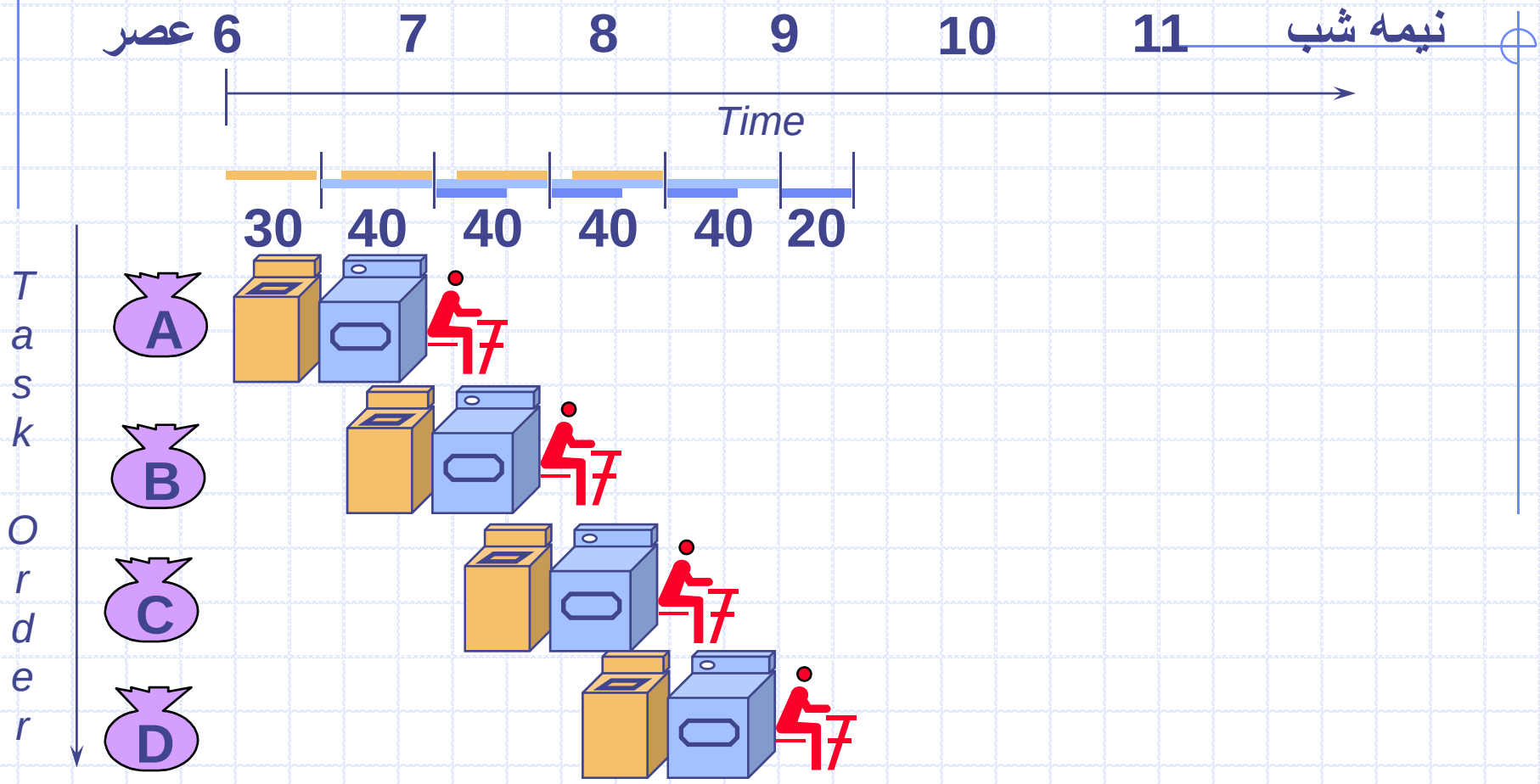
- خط لوله ای کردن، روشی را برای اجرای همزمان چند دستورالعمل بوجود می آورد.
- مثال شستن لباسها
- Ann,Brian,Dave,Cathy هر کدام توده ای از لباس را برای شستن، خشک کردن و تا کردن در اختیار دارند.
- شستن 30 دقیقه زمان می برد.
- خشک کن 40 دقیقه زمان می برد.
- تا کردن 20 دقیقه زمان نیاز دارد.

شستن ترتیبی



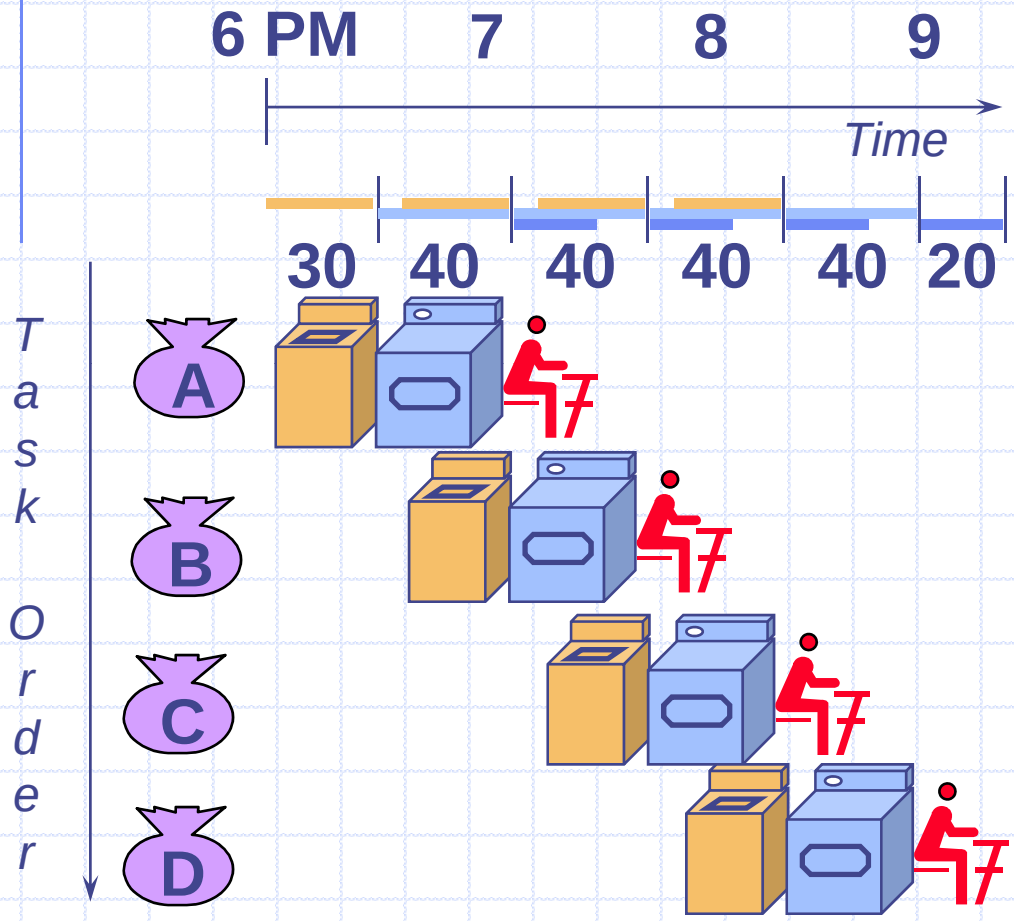
- شستن ترتیبی 4 توده لباس، 6 ساعت زمان می برد.
- اگر آنها خط لوله ای را فرا گرفته بودند، چقدر طول می کشید؟

شستن خط لوله ای شده:



- شستن خط لوله ای 4 توده لباس،3 ساعت و نیم زمان می برد.

دروس خط لوله ای کردن



- خط لوله ای به زمان تاخیر یک کار کمکی نمی کند، بلکه توان عملیاتی کل بار کاری را بهبود می بخشد.
- نرخ خط لوله محدود به کندترین مرحله است.
- اجرای همزمان چند کار نیاز به منابع متفاوتی دارد.
- پتانسیل تسریع = تعداد مراحل خط لوله
- طول نامتوازن مراحل خط لوله باعث کاهش تسریع می شود.
- زمان مصرفی برای پر شدن و خالی شدن خط لوله باعث کاهش تسریع می شود.
- متوقف شدن به خاطر وابستگی ها(STALL)

مفهوم پایه

- خط لوله: چندین دستورالعمل به طور همزمان در حال اجرایند.
- خط لوله به بخشها یا قطعات تقسیم می شود.
- چرخه ماشین:
  - زمان مورد نیاز برای گذر از یک مرحله
  - چرخه ماشین بوسیله کندترین مرحله خط لوله معین می گردد.
  - معمولا چرخه ماشین = چرخه ساعت

- در یک ماشین خط لوله ای شده کاملا متوازن:
  - زمان $= \frac{\text{زمان غیر خط لوله ای}}{\text{تعداد بخشهای خط لوله}}$ (1)
- در یک ماشین معمولی

(1) درست نیست،

زمان مراحل برابر نیست، یک سربار وجود دارد.

اما می تواند تا اختلاف 10% به (1) نزدیک شود.

خط لوله ای کردن به عنوان یک تکنیک معماری

- بطور کلی، تقریبا برای کاربر نامحسوس است.
- Scalar pipelined machine vs. vector machine

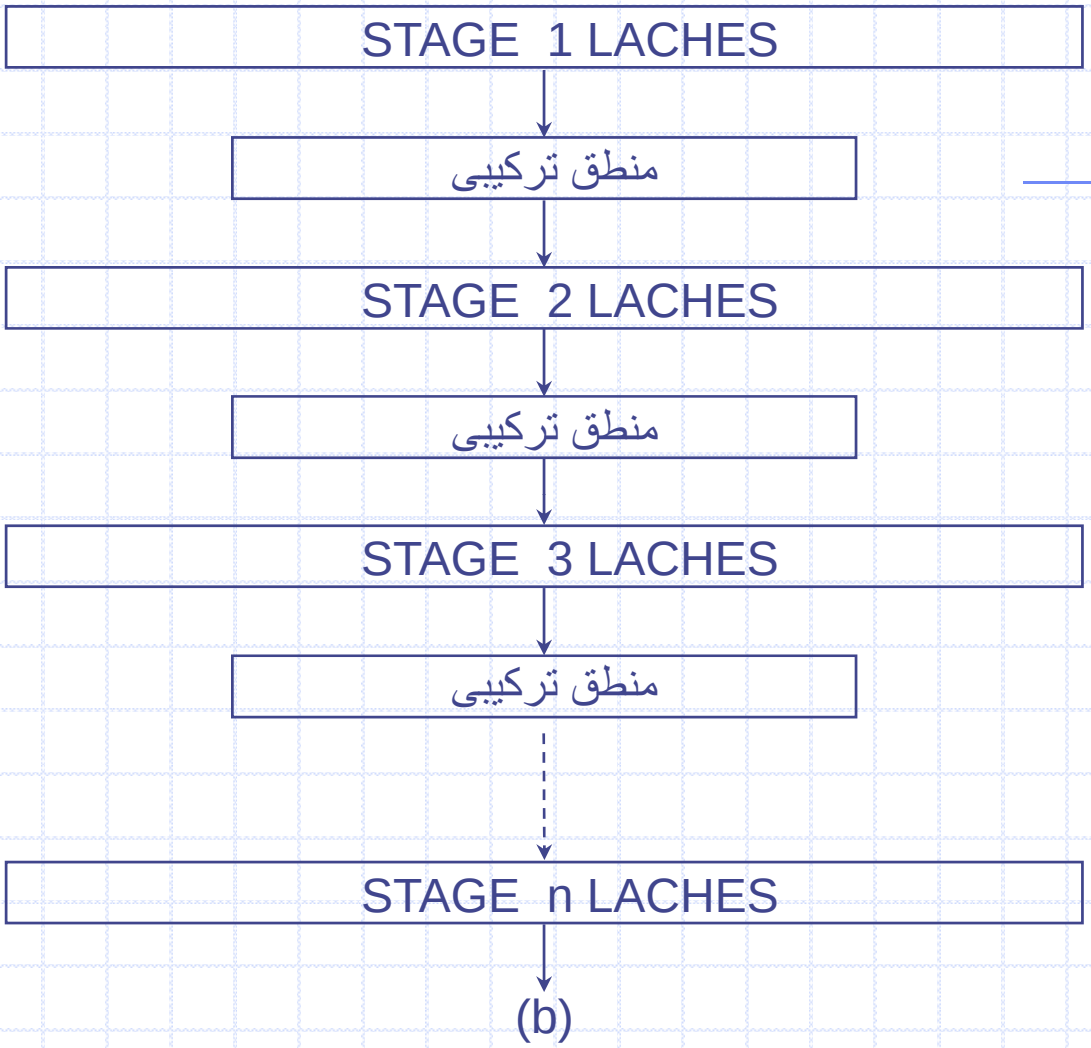
- توان عملیاتی خط لوله ای :

$$\frac{\text{تعداد دستورالعملهای کامل شده}}{\text{چرخه}}$$

- بنابراین خط لوله ای کردن توان عملیاتی(ظرفیت پذیرش) را افزایش می دهد، اما زمان اجرای هر دستورالعمل بدون تغییر می ماند.
- نرخ ساعت یک ماشین خط لوله ای شده محدود شده است به:
  - latch time
  - clock skew
  - زمان تاخیر مورد نیاز برای انتشار سیگنال ساعت در یک تراشه

محدودیتهای کارایی در یک خط لوله

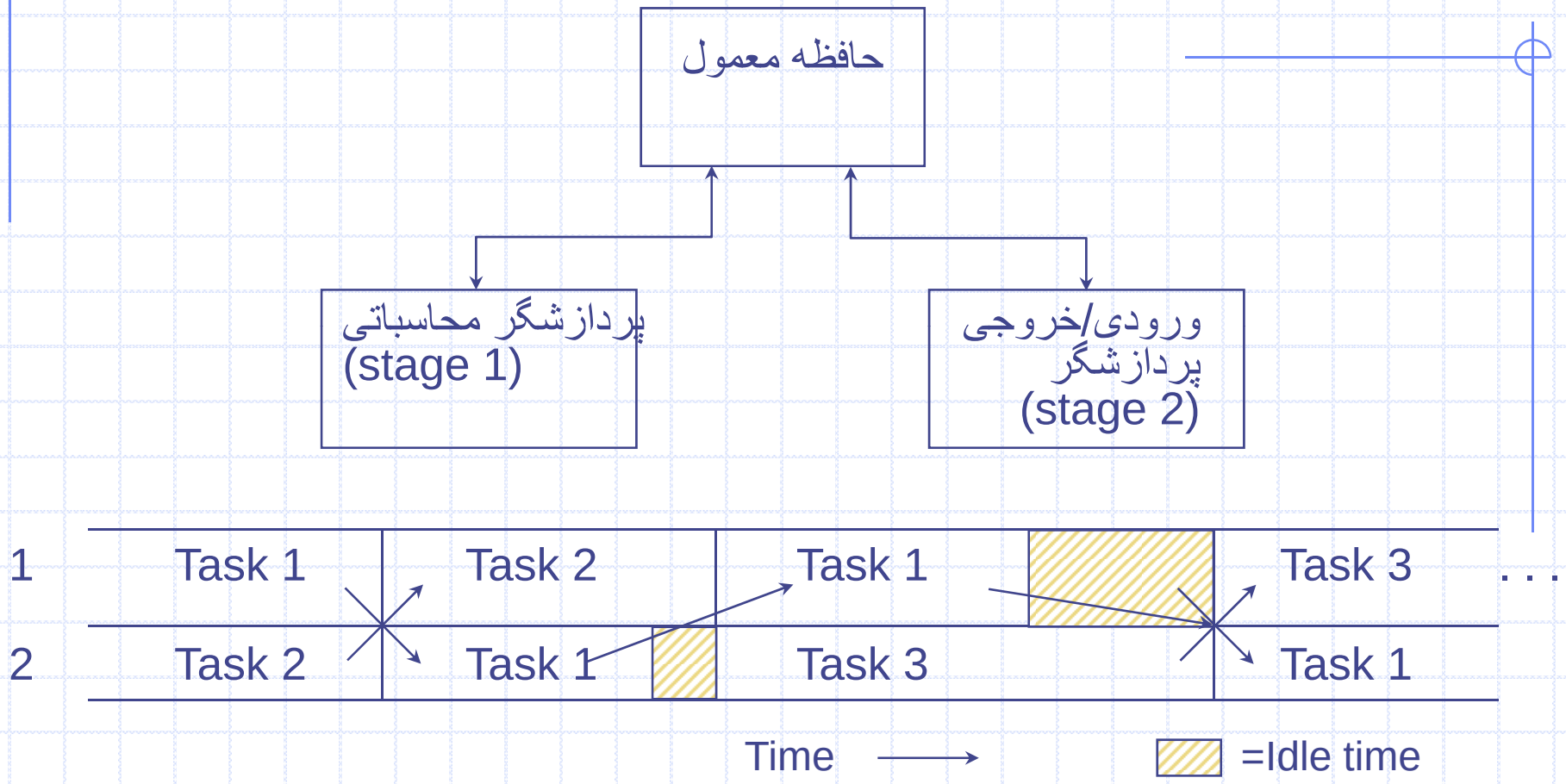
- نمی توان از کندترین مرحله انتظار سرعت داشت
- پیچیدگیها در واقعیت، زمانهای پردازش متفاوت برای مراحل مختلف، فعل و انفعالات/وابستگیها بین مراحل – ممکن است باعث وابستگی داده ای شود. (پویا)



فاکتورهای مهم در مسیر لوله(مدت چرخه) ، تاخیر latch و clock skew

رویهم اندازی(Overlap) در برابر خط لوله ای

- خط لوله ای
  - ارتباط قوی زیر بخشها(sub functions)
  - زمان پایه ثابت مرحله
  - ارزیابی تابع پایه مستقل
- مشترکات
  - پیوند ضعیف زیر تابع ها
  - زمان متغیر مرحله
  - ارزیابی وابستگی بین تابعی



رویهم اندازی CPU/I/O

خط لوله پویا و ایستا

- **ایستا:**
  - تنها ارزیابی های مکرر همان کار با داده های متفاوت انجام می شود
  - no dynamic data dependencies between initiations -
  - طرح ثابت آغاز ها
- **پویا:پویا متضاد است با ایستا**
  - (اسنکرون)
  - (روی هم افتادگی)
- **مثالی از خط لوله ایستا:**
  - جمع ممیز شناور در خط لوله

یک تابع واحدلوله_ چند تابع لوله

- **یک تابع واحد لوله**
  - مثال:جمع لوله
- **چند تابع لوله**
  - مثال
  - لوله حسابی
- **بردار لوله**
  - کنترل لوله قابل برنامه ریزی
  - (دستورالعمل برداری وظیفه و ورودی ها را تعیین می کند)

خط لوله MIPS

- **مراحل خط لوله:**
  - واکشی
  - ID(کد برداری +واکشی ثبات ها)
  - محاسبات
  - دستیابی به حافظه
  - باز نویسی

Instruction number	Clock number 1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

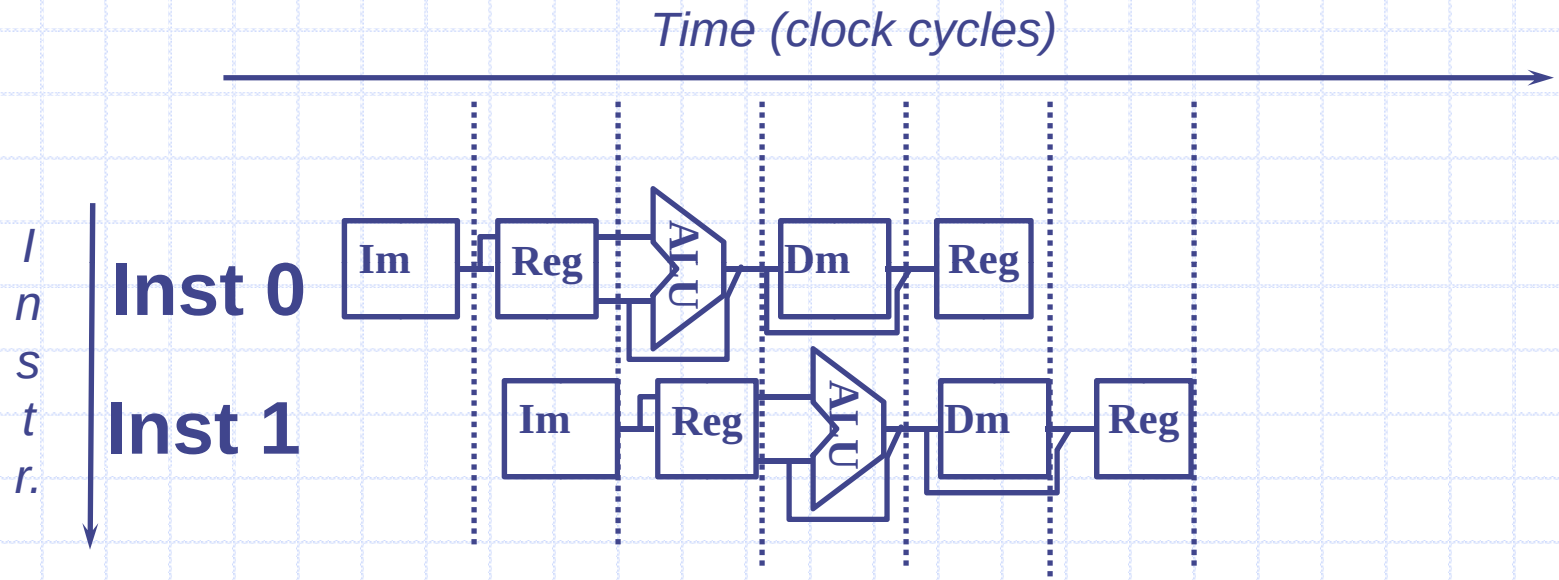
در هر چرخه ساعت دستورالعمل دیگری واکشی شده و اجرا در 5 مرحله آغاز می شود.اگر یک دستورالعمل شروع شود در هرچرخه ساعتstarted every clock cycle, the performance will be five times that of a machine that is not pipelined.

خط لوله MIPS

نمایش دیگر

	1	2	3	4	5	6	7	8	9
IF	1	2	3	4	5				
ID		1	2	3	4	5			
EX			1	2	3	4	5		
MEM				1	2	3	4	5	
WB					1	2	3	4	5

نمایش منقوش خط لوله

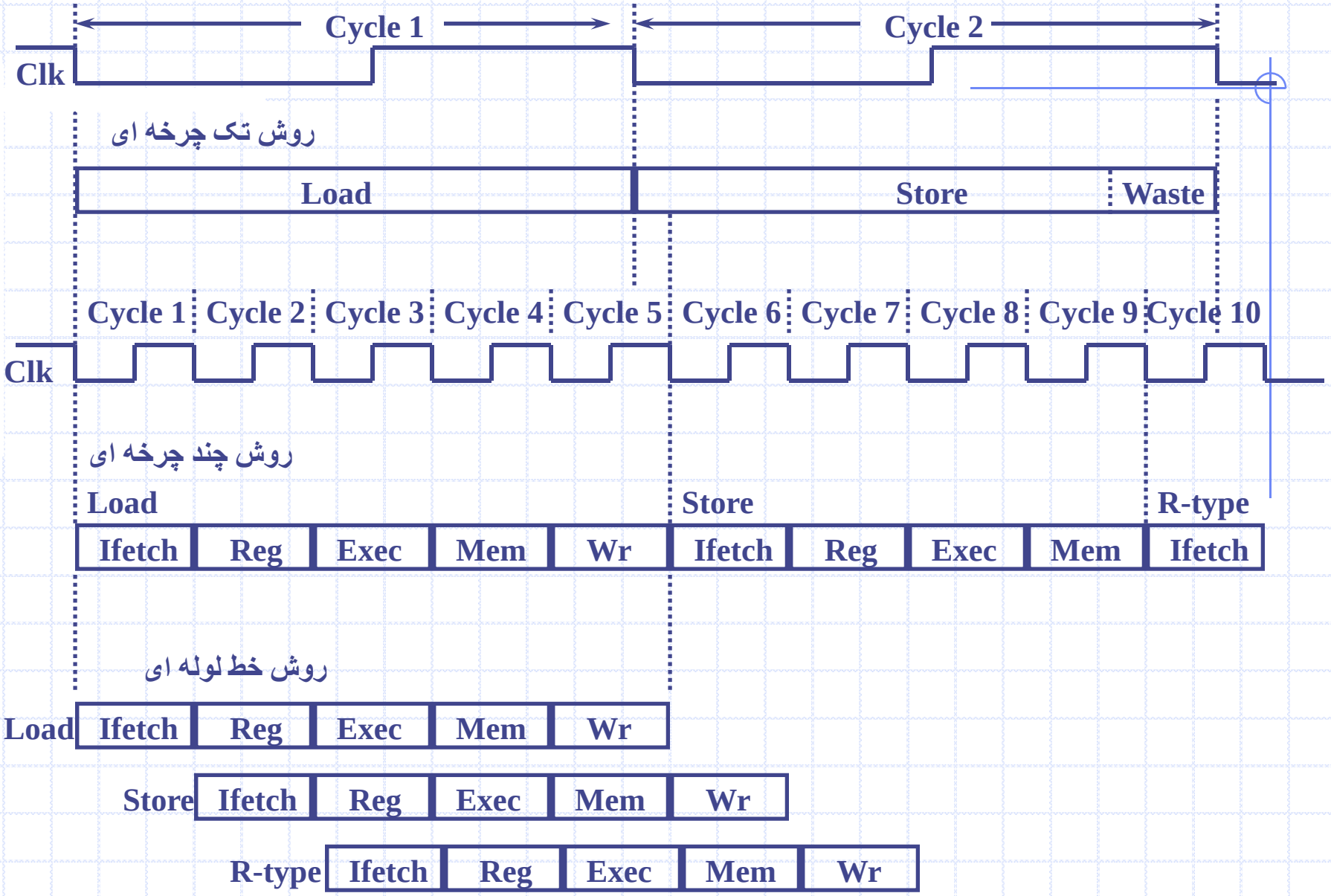


- با پاسخ دادن به سوالات مشابه می تواند کمک کند
  - چند چرخه زمان می برد که این کد اجرا شود؟
  - ALU در طول 4 چرخه چه کاری انجام می دهد؟
  - آیا 2 دستورالعمل سعی در استفاده همزمان از یک منبع مشترک دارند؟

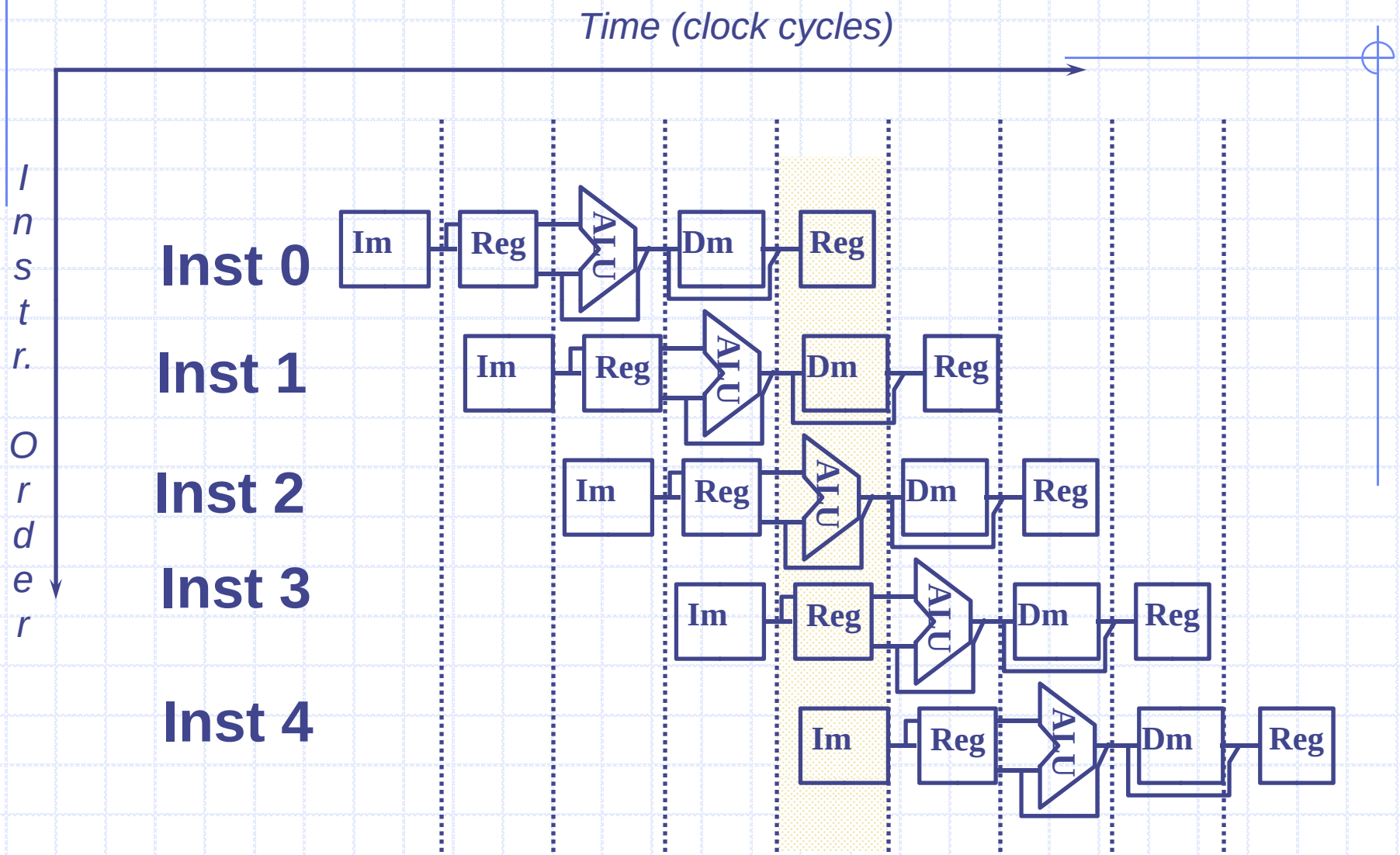
چرا خط لوله ای؟

- فرض کنید
  - 100 دستورالعمل را می خواهیم اجرا کنیم
  - ماشین تک چرخه ای که یک چرخه ساعت دارد NS45 زمان می خواهد
  - ماشین چند چرخه ای و خط لوله ای که چندچرخه زمان می برند NS10زمان می خواهد
  - CPIماشین چند چرخه ای 4.6است
- ماشین تک چرخه ای
  - 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- ماشین چند چرخه ای
  - 10 ns/cycle x 4.6 CPI x 100 inst = 4600 ns
- ماشین خط لوله ای ایده ال
  - 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns
- میزان تسریع خط لوله ایده آل و ماشین تک چرخه ای
  - 4500 ns / 1040 ns = 4.33
- What has not yet been considered?

تک چرخه ای_ چند چرخه ای _خط لوله ای



چرا خط لوله؟ برای اینکه منابع را در اینجا داریم!



آیا خط لوله می تواند برای ما مشکل ایجاد کند؟

- بله: هزاردهای خط لوله ای
  - هزاردهای ساختاری: تلاش برای استفاده از یک منبع یکسان برای 2 کار متفاوت در یک زمان
    - مثال: سعی در خواندن همزمان دو دستورالعمل از یک حافظه
  - هزاردهای داده ای: تلاش برای استفاده از یک آیتم قبل از اینکه آماده شود
    - دستورالعمل وابسته است به نتیجه دستورالعمل قبلی که هنوز در خط لوله است

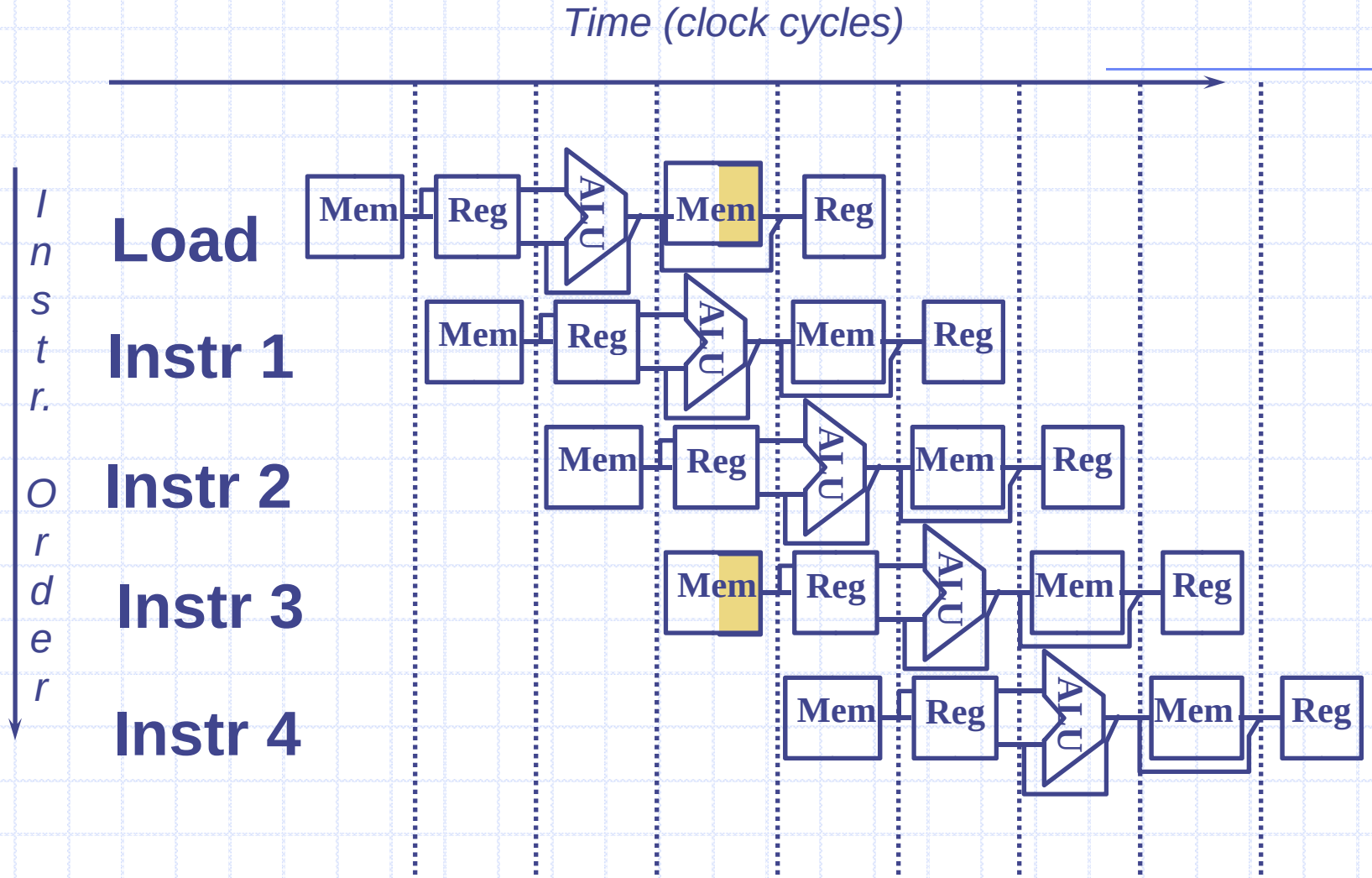
```
add r1, r2, r3
sub r4, r2, r1
```

  - هزاردهای کنترلی: تلاش برای ظاهر کردن یک تصمیم قبل از اینکه شرط اجرا شود
    - دستورالعمل های انشعاب

```
beq r1, loop
add r1, r2, r3
```

- همیشه میتوان هزاردها را با انتظار رفع کرد
  - کنترل خط لوله ای باید هزارد را کشف کند
  - برداشتن عمل(تاخیر عمل) می تواند هزارد را حل کند

حافظه تک یک هزارد ساختاری است



تشخیص در این نمونه آسان است(نیمه روشن سمت راست یعنی خواندن و نیمه چپ یعنی نوشتن)

هزاردهای ساختاری کارایی را محدود می کند

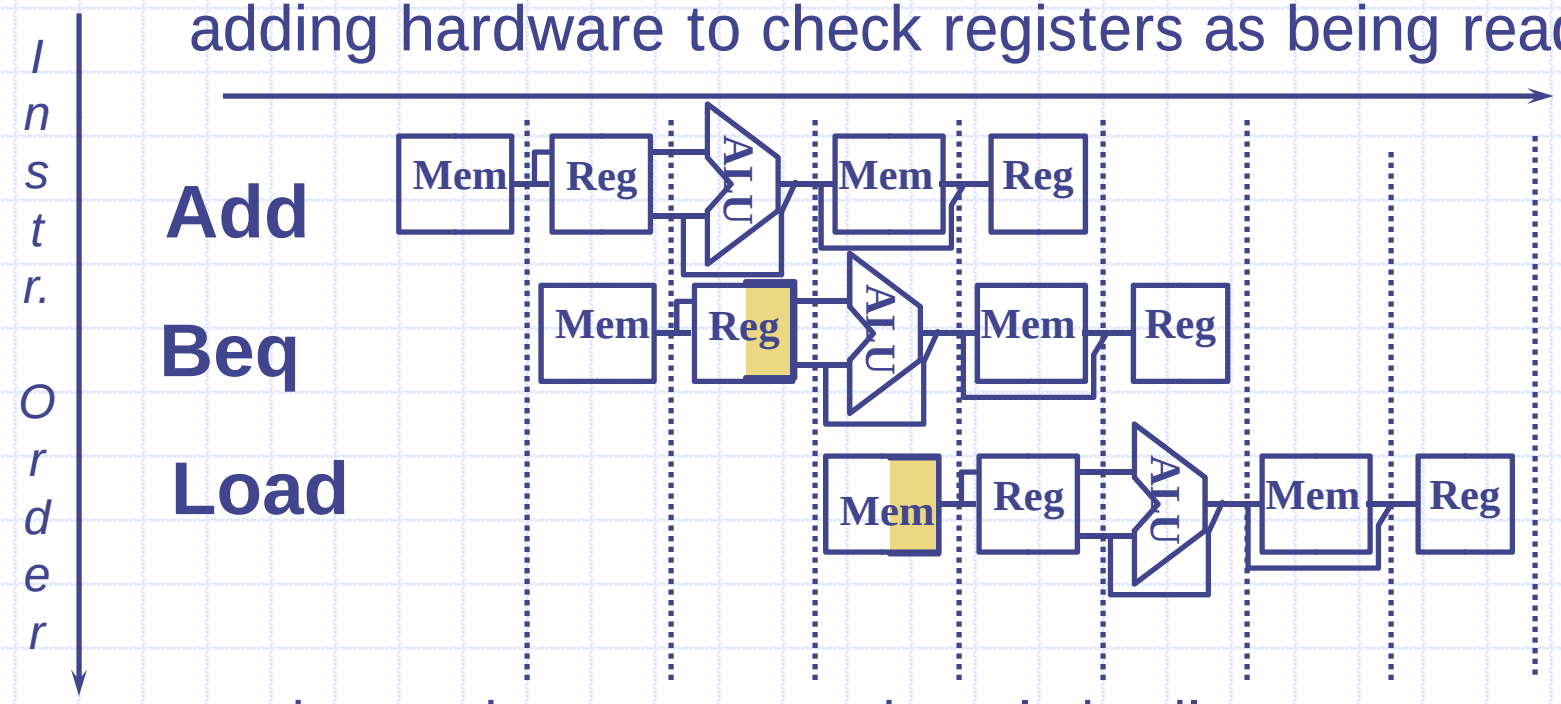
- مثال:اگر 1.3 حافظه در هر دستورالعمل دستیابی شود و فقط یک حافظه مورد دستیابی قرار می گیرد در هر چرخه

CPI = 1.3میانگین

  - منبع بیشتر از 100%مورد استفاده واقع شده
- راه حل اول: جدا کردن حافظه داده و دستورالعمل از یکدیگر
- راه حل دوم: اجزه دهیم حافظه بیش از یک خواندن و نوشتن در یک چرخه انجام دهد
- راه حل سوم: توقف

راه حل های هزارد کنترلی

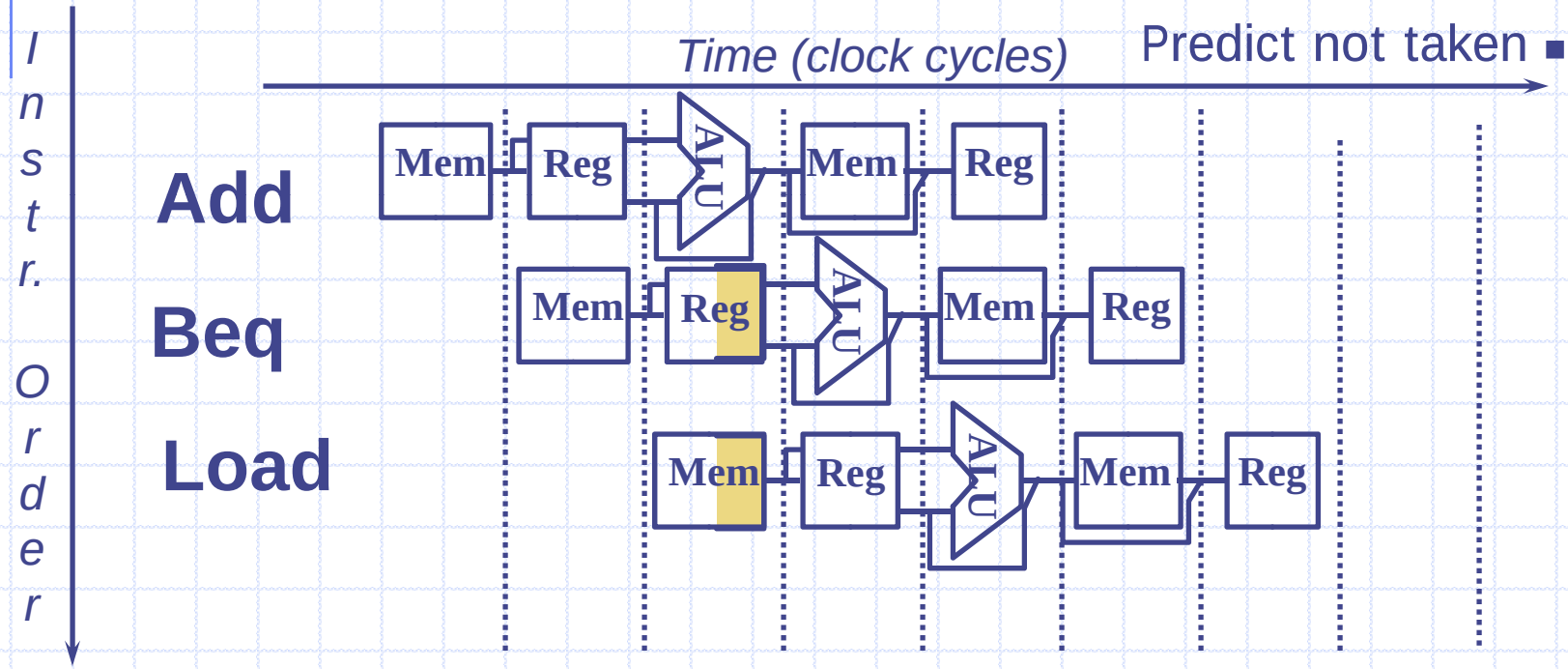
- توقف:تا زمانی که تصمیم مشخص شود
- Its possible to move up decision to 2nd stage by adding hardware to check registers as being read



برخورد :هر دستورالعمل انشعاب دو چرخه ساعت زمان می برد=>کند شدن

راه حل های هزارد کنترلی

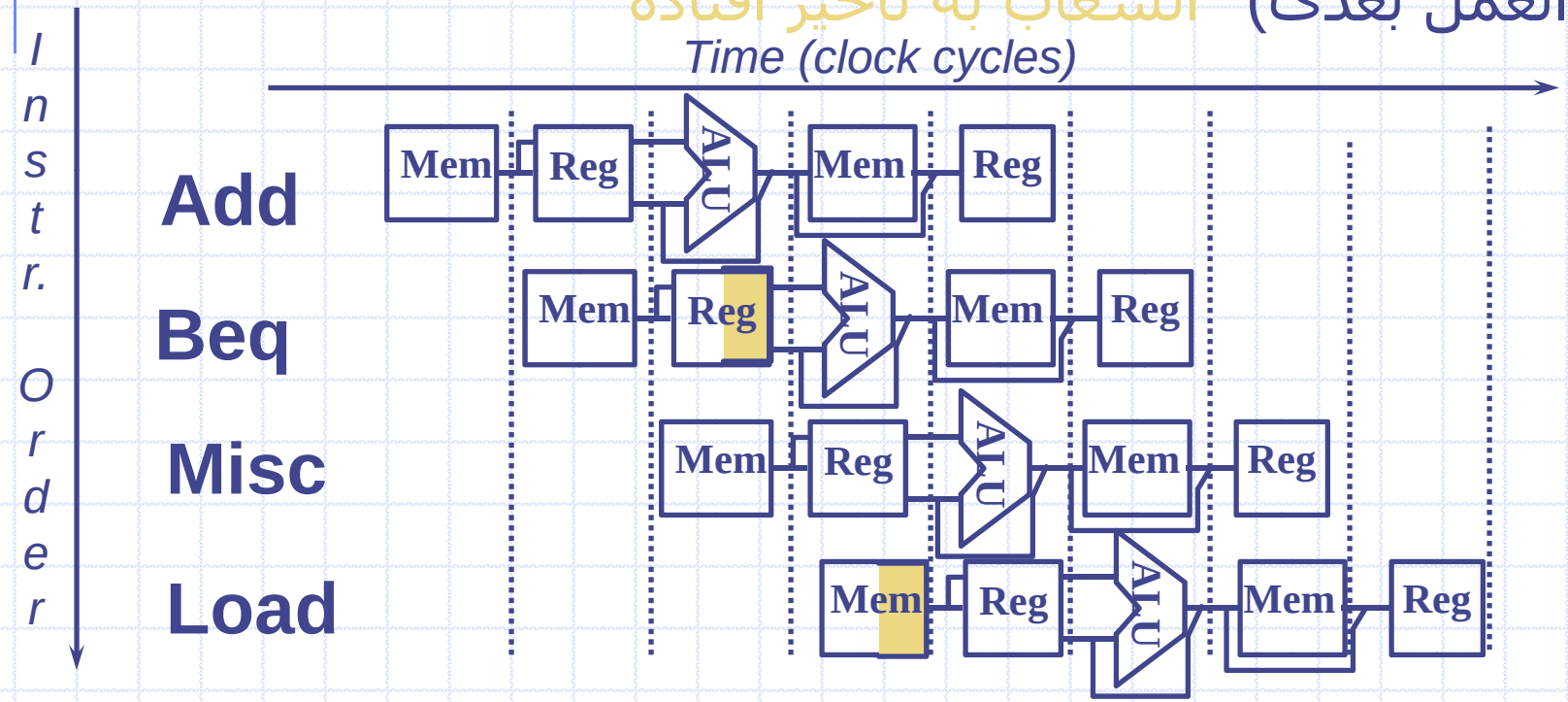
- پیشگویی کردن:یک مسیر حدس زدن بعد تهیه پشتیبان ذ اگر اشتباه بود



- برخورد: هر دستورالعمل انشعاب اگر درست باشد یک چرخه ساعت و اگر اشتباه باشد 2 چرخه زمان می برد(50% موارد درست است)
- More dynamic scheme: history of 1 branch (- 90%)

راه حل های هزارد کنترلی

- دوباره تعریف کردن رفتار انشعاب(اتفاق می افتد بعد از دستورالعمل بعدی) "انشعاب به تاخیر افتاده"



- Impact: 1 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)
- Launch more instructions per clock cycle=>less useful

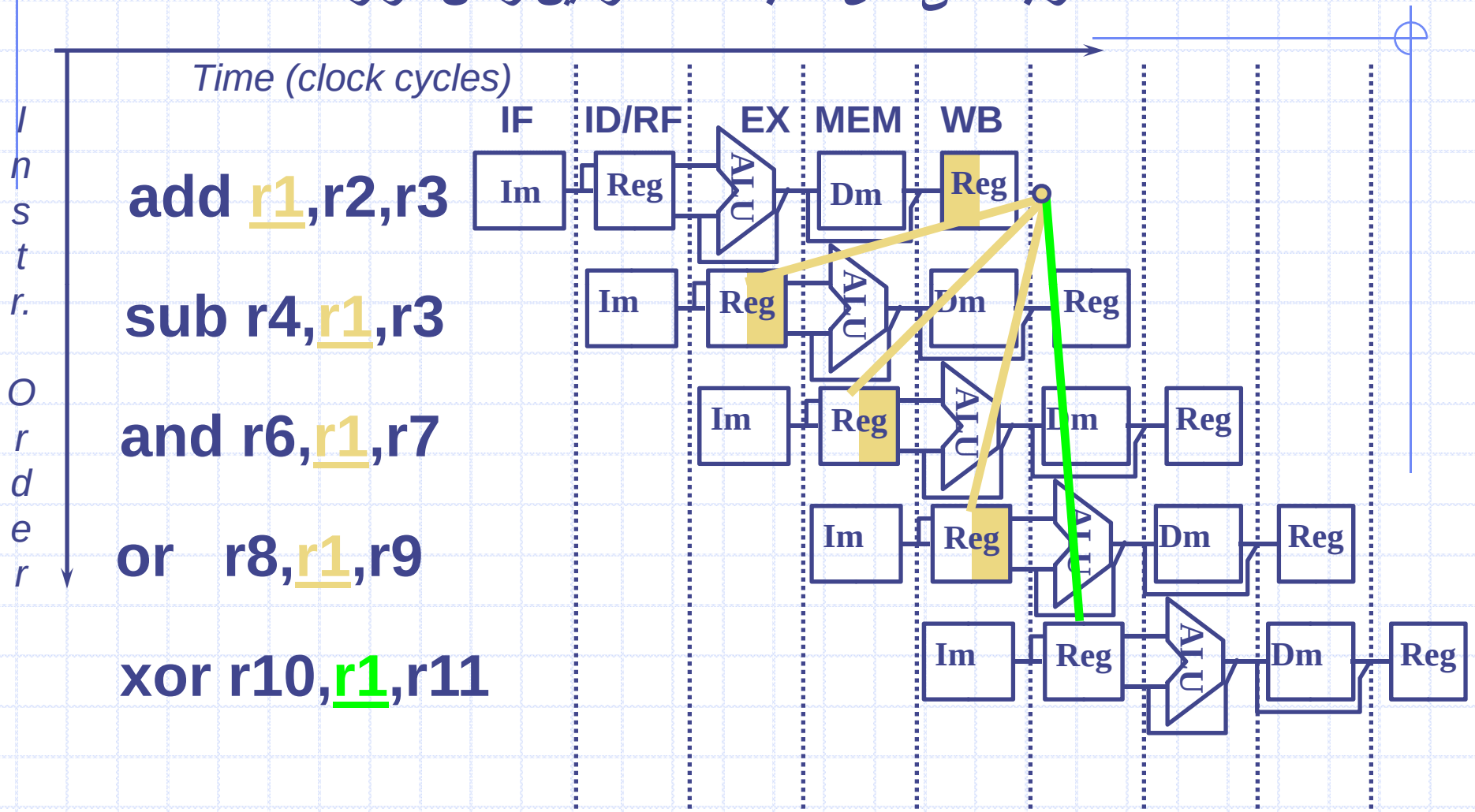
هزاردهای داده ای روی r1

مشکل:r1نمی تواند به وسیله دستورالعمل های دیگر خوانده شود قبل از اینکه به وسیله add نوشته شود

```
add r1 ,r2,r3
sub r4, r1 ,r3
and r6, r1 ,r7
or  r8, r1 ,r9
xor r10, r1 ,r11
```

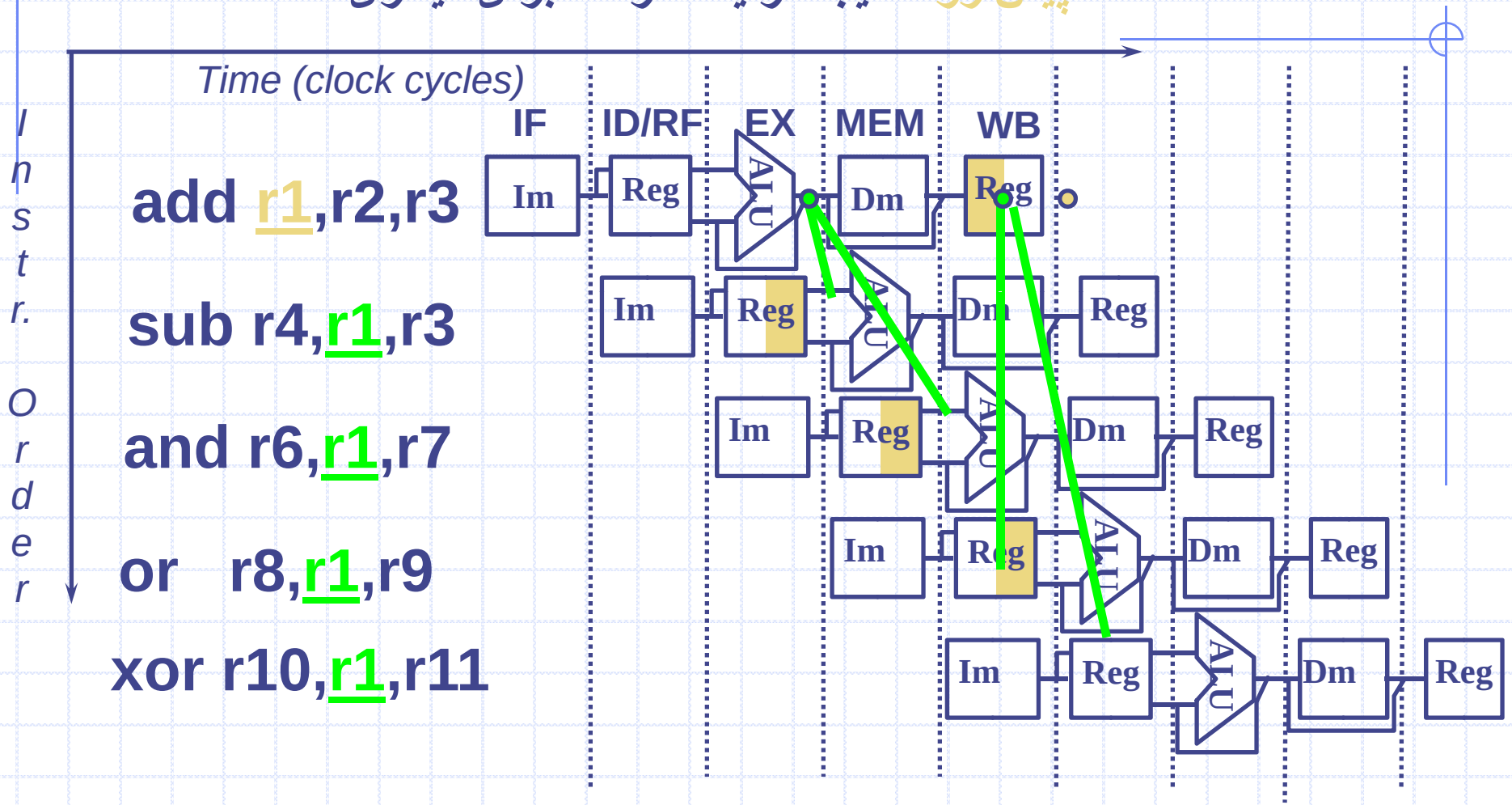
هزارد داده ای روی r1:

- وابستگی های عقب افتاده در این زمان هزاردها هستند



راه حل هزارد داده ای:

- "پیش رو" نتیجه از یک مرحله برای دیگری

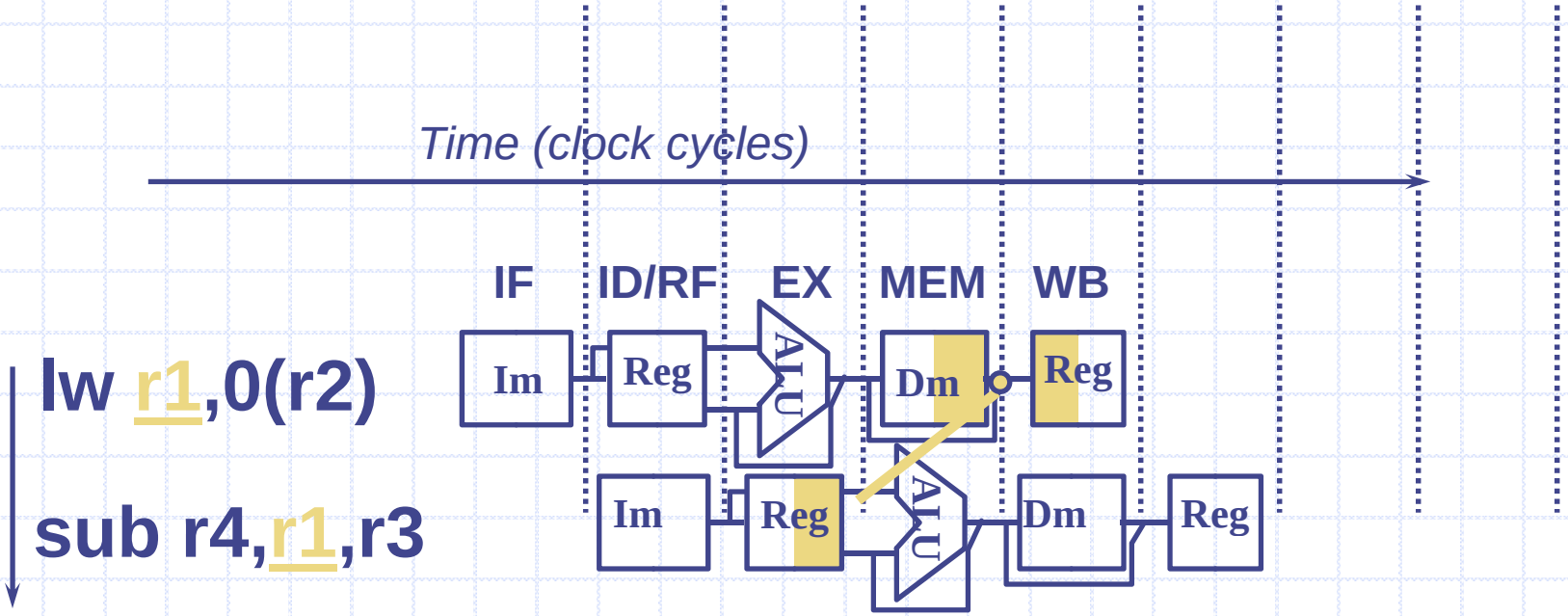


- "or" بسیار خوب اگر که تعریف نوشتن/خواندن درست است

Forwarding(یاگذشتن): در مورد بار کردن چی؟

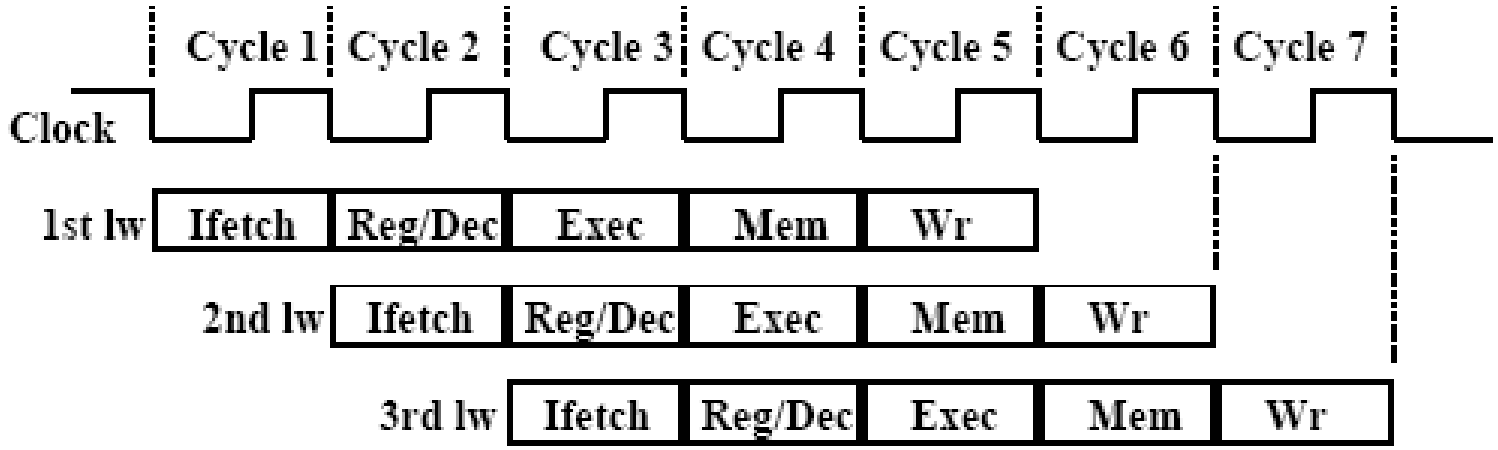
- وابستگی های به جا مانده از قبل هزارد هستند

- نمی توان با ارسال(Forwarding)این مسئله را حل کرد؟
- باید دستورالعمل وابسته به بارگذاری را توقف /تاخیر دهیم



طراحی یک مسیر داده خط لوله ای شده

خط لوله ای کردن دستورالعمل بارکردن



- 5 تا واحد تابعی مستقل در مسیر داده خط لوله ای هستند.

ًٌٍَُِّ - حافظه دستورالعمل برای مرحله واکشی

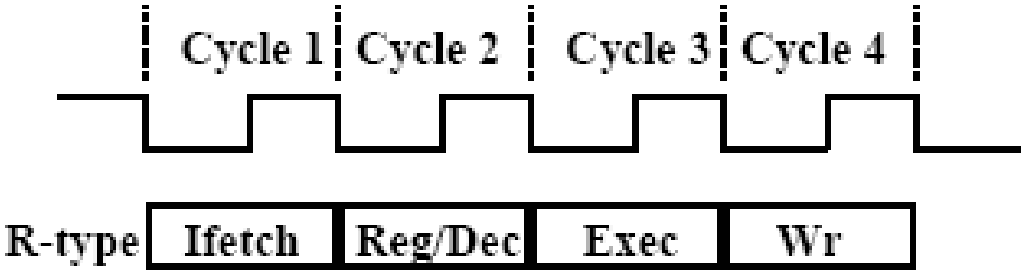
-خواندن از بانک ثباتی برای مرحله Reg/Dec (bus A and busB)

- ALUبرای مرحله اجرا

-حافظه داده برای مرحله دسترسی به حافظه

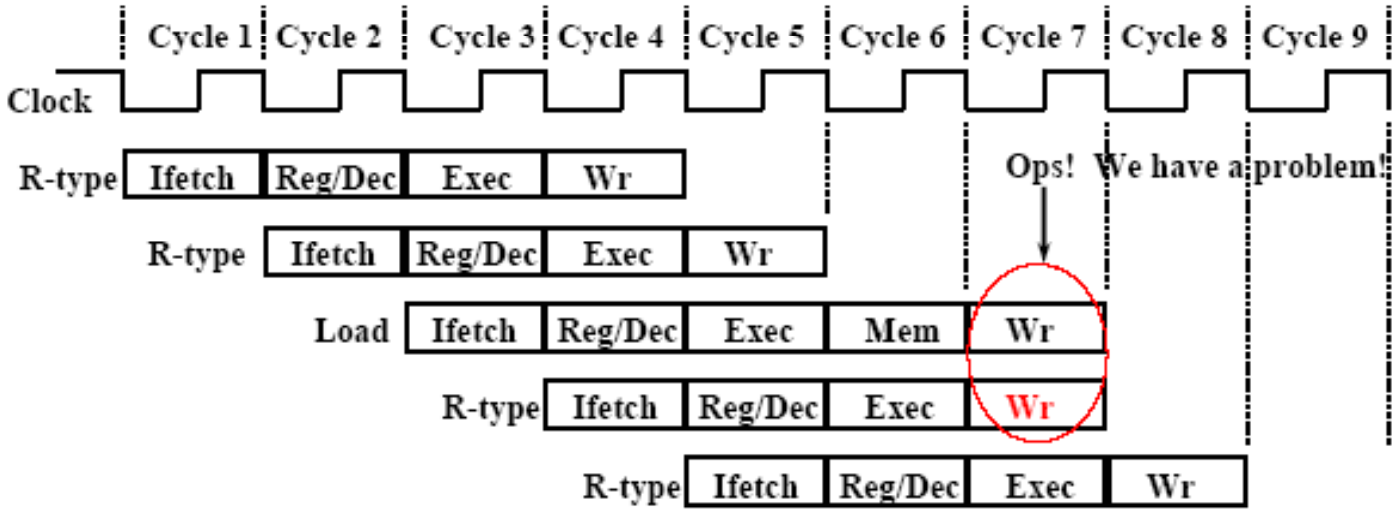
-نوشتن بر بانک ثباتی برای مرحله Wr (bus W)

4 مرحله برای R-type



- واکشی:واکشی دستورالعمل
  - واکشی دستورالعمل از حافظه دستورالعمل
  - جدید کردنPC
- Reg/Dec: واکشی ثباتها و کد برداری دسترالعمل ها
- Exec: عمل کردن ALUبر روی 2 تا ثبات عملوند
- Wr: باز نویسی خروجی ALUبر روی بانک ثباتی

خط لوله ای کردن دستورالعمل های R-typeوLoad



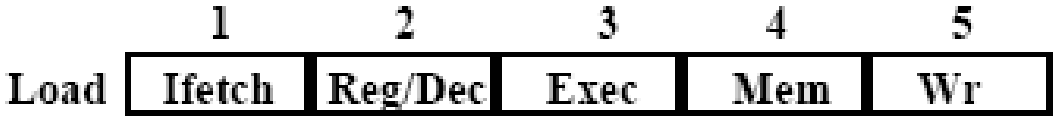
ما در خط لوله ای تضاد داریم یا هزارد های ساختاری

-دو دستورالعمل سعی در نوشتن در بانک ثباتی به صورت همزمان دارند!

-تنها نوشتن یک پورت

مشاهدات مهم

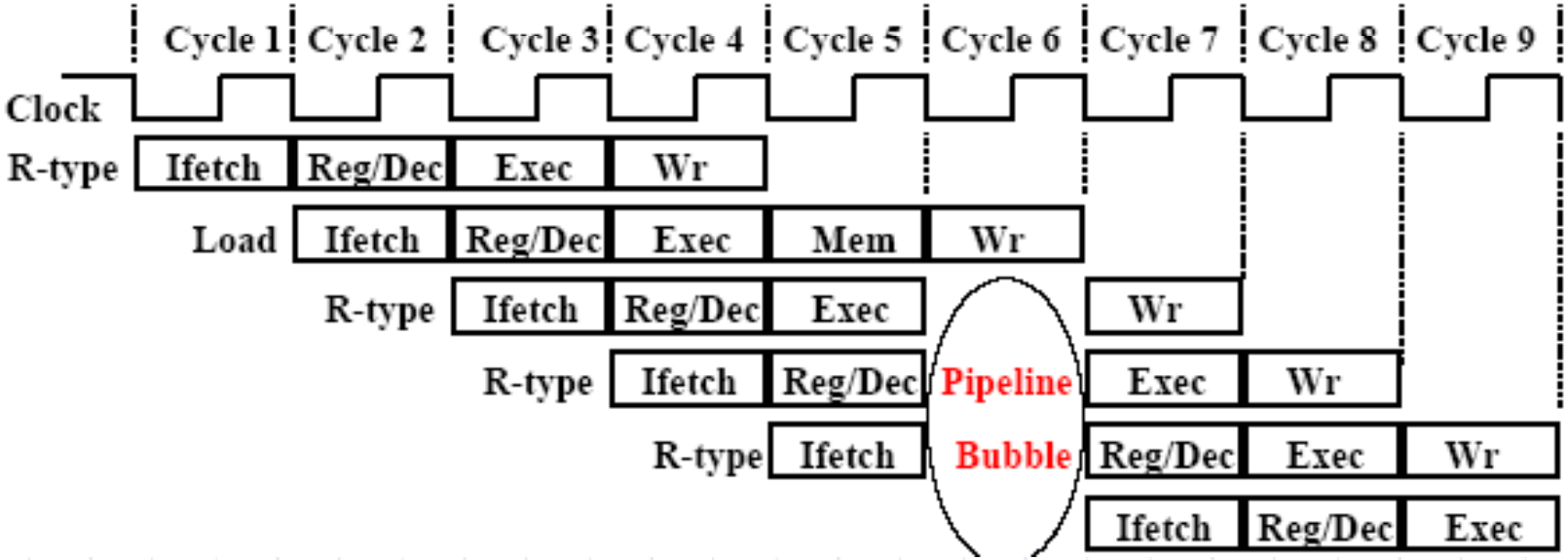
هر واحد کاری تنها یک بار برای هر دستورالعمل می تواند استفاده شود
-هر واحد کاری باید در همان مرحله برای همه دستورالعمل ها استفاده شود:
-در دستورالعمل Load نوشتن در بانک ثباتی در مرحله پنجم انجام می شود



-در دستورالعملR-type نوشتن در بانک ثباتی در مرحله چهارم انجام می شود

2 راه حل برای هزارد خط لوله ای وجود دارد

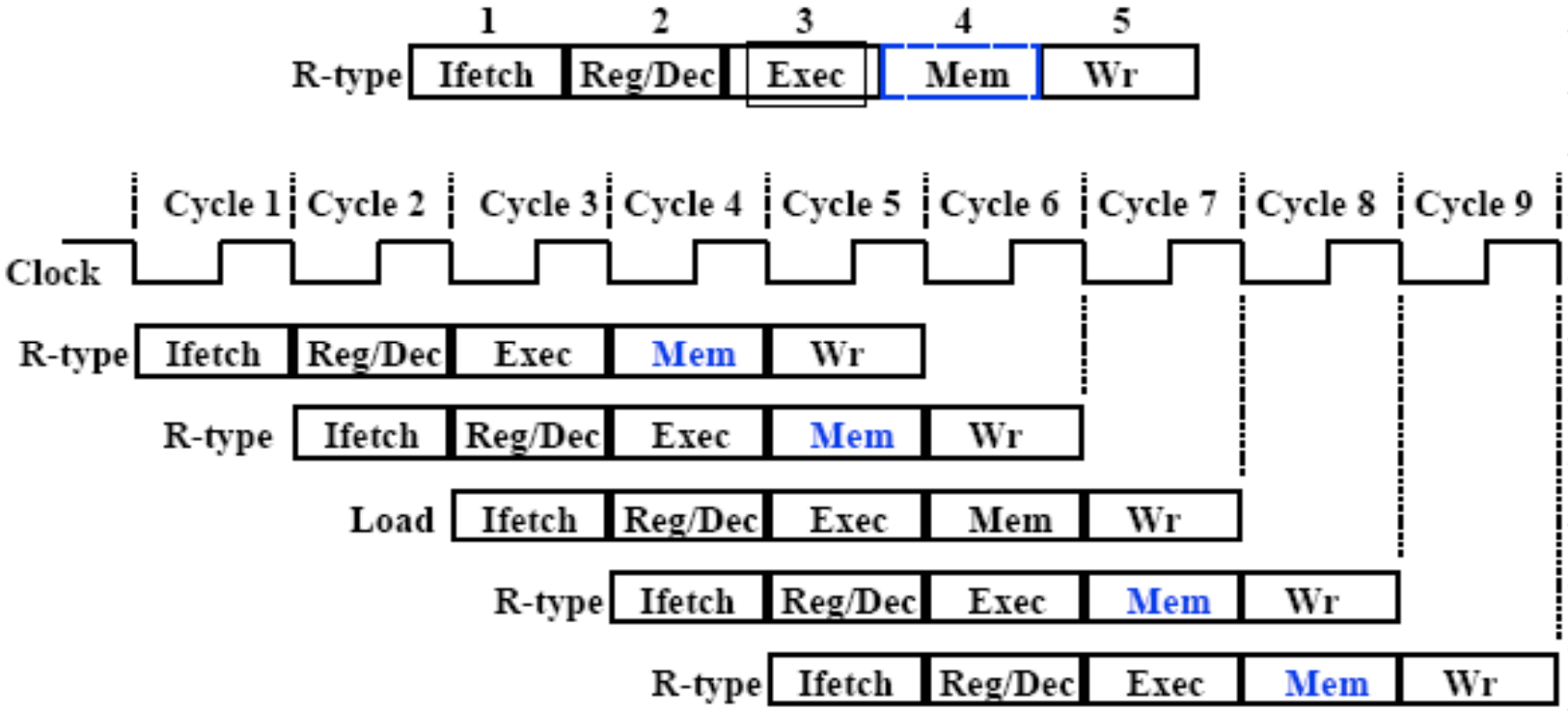
راه حل اول :وارد کردن"حباب"در خط لوله



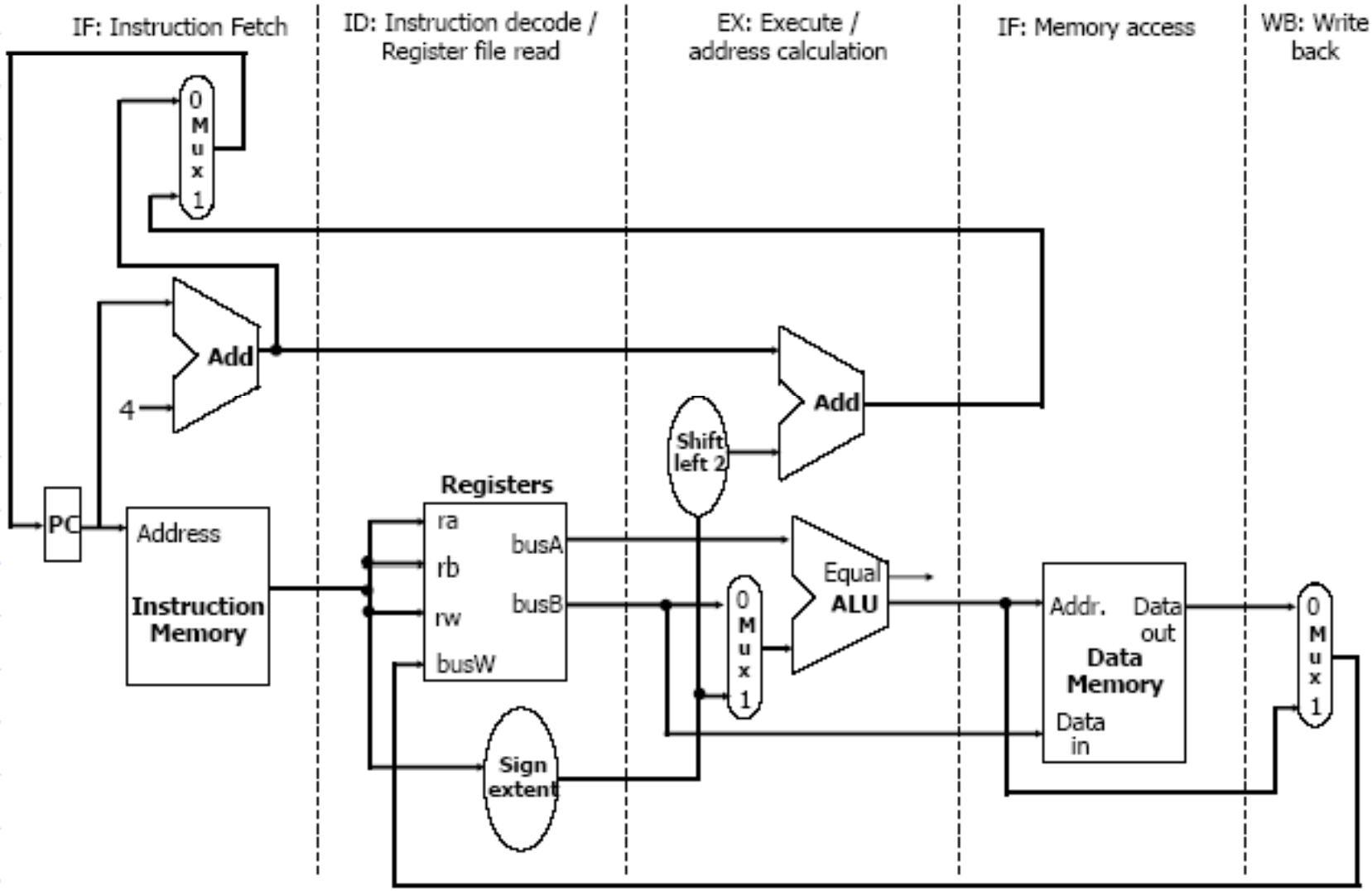
- وارد کردن حباب در خط لوله برای مانع شدن از دو نوشتن همزمان در یک چرخه
  - منطق کنترل می تواند پیچیده شود.
  - از دست دادن واکشی دستورالعمل و فرصت انتشار.
- هیچ دستورالعملی در چرخه 6 شروع نمی شود

راه حل دوم : تأ خیر نوشتن R-typeبه وسیله یک چرخه

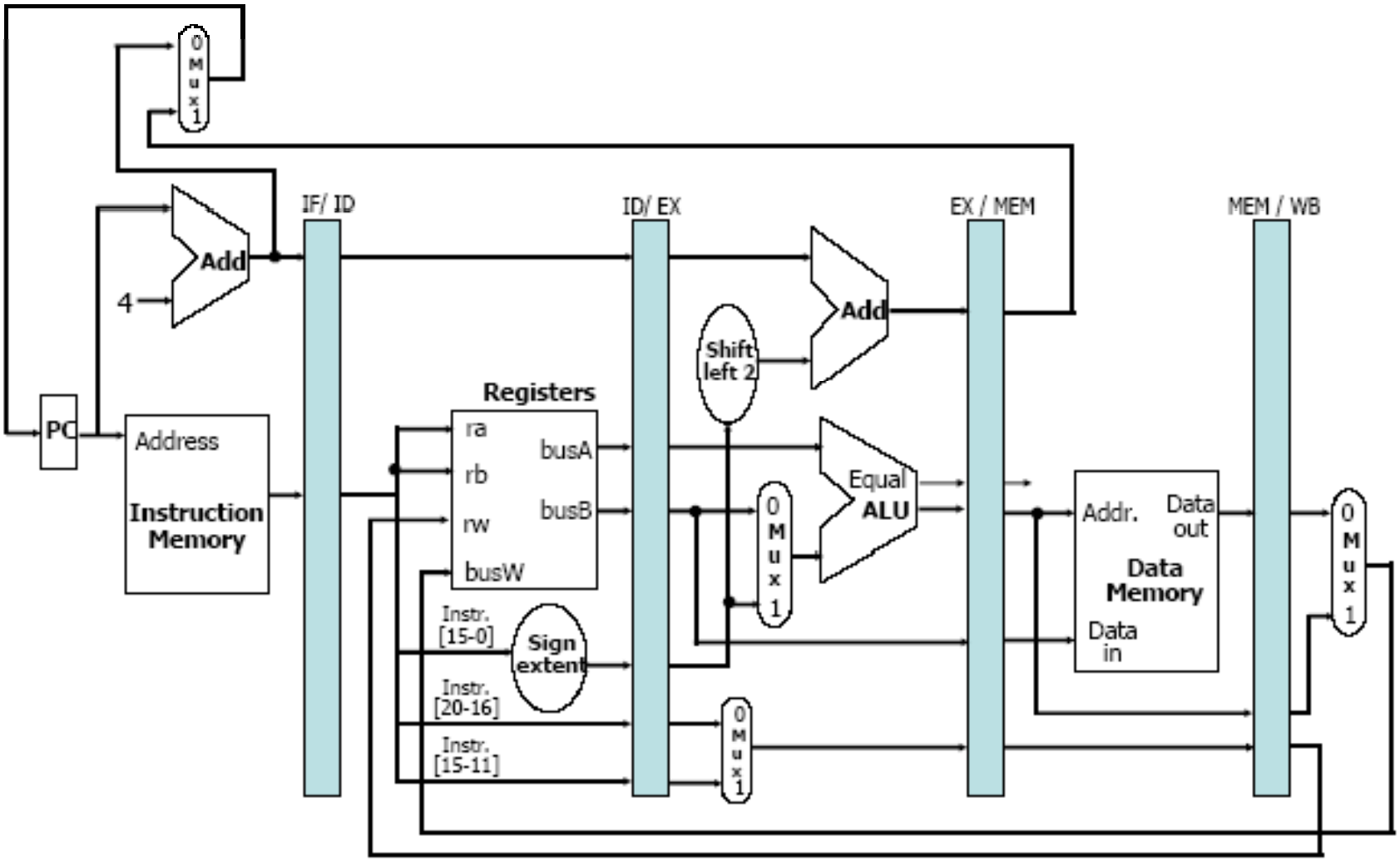
- تأ خیر نوشتن R-typeبه وسیله یک چرخه
- - حال می توان نوشتن دستورالعمل های R-type را در مرحله پنجم انجام داد
- مرحله دستیابی به حافظه یک مرحله بلااستتفاده است:هیچ چیزی در آنجا انجام نمی شود.



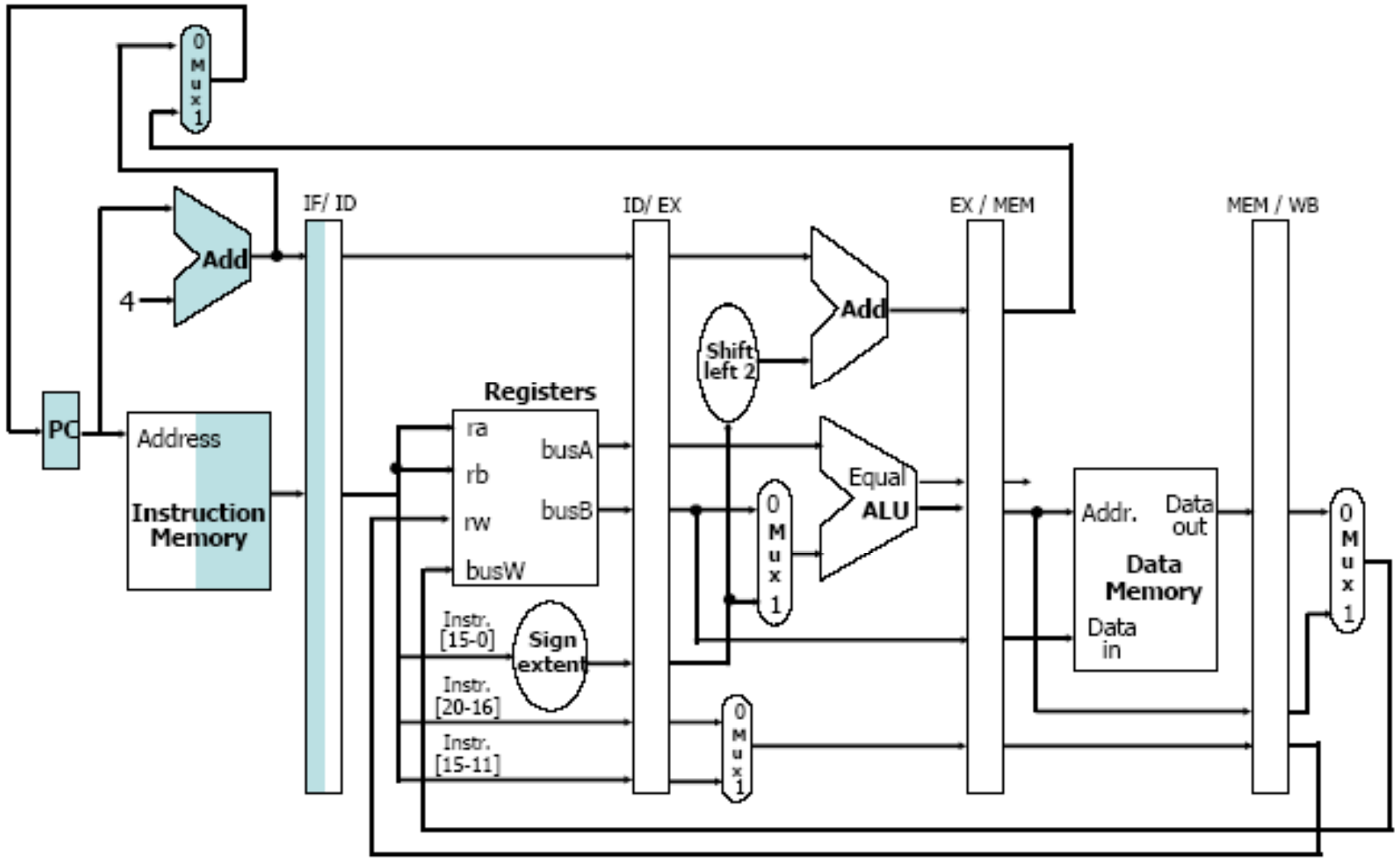
مسیر داده تک چرخه ای



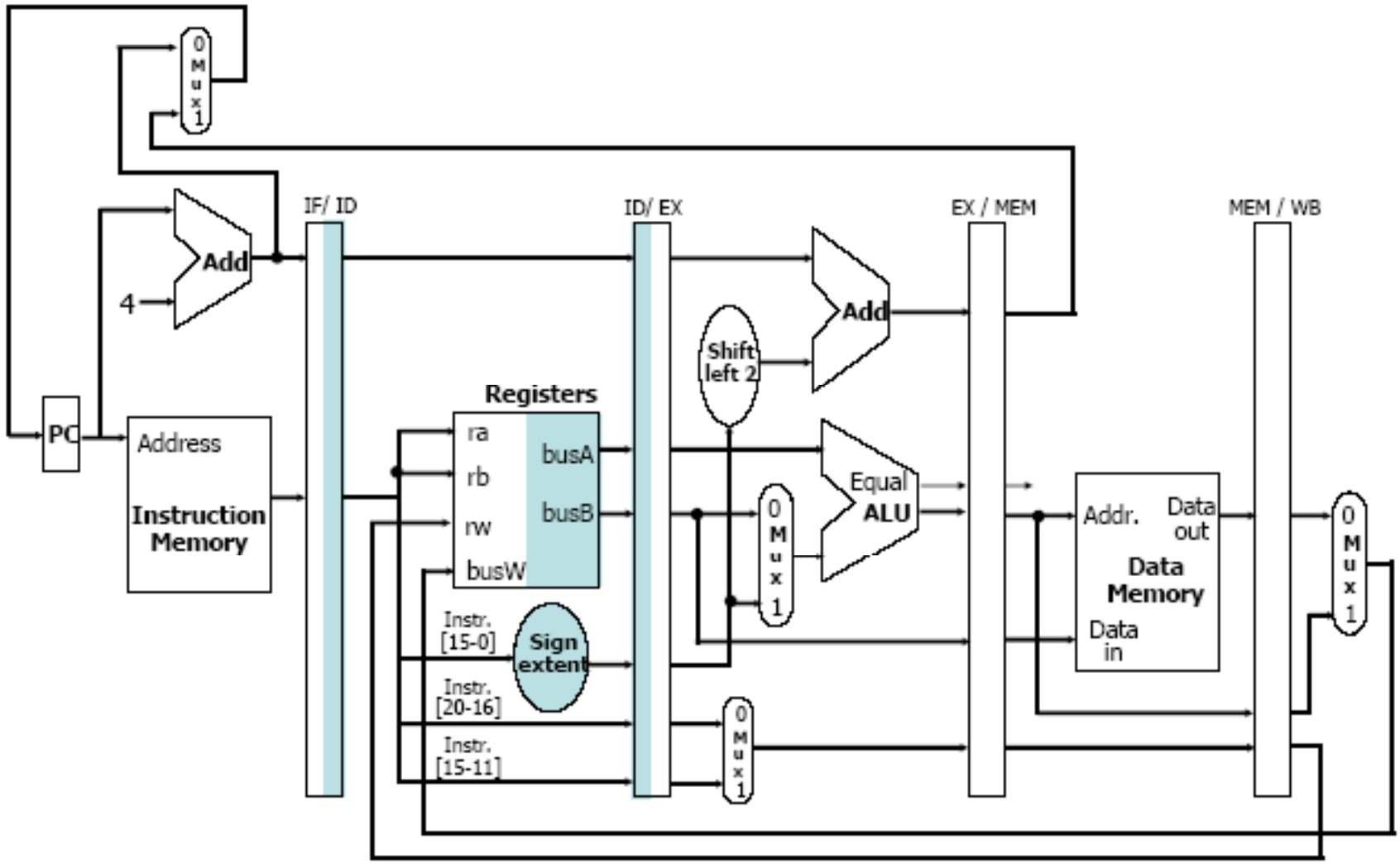
نسخه خط لوله ای شده مسیر داده



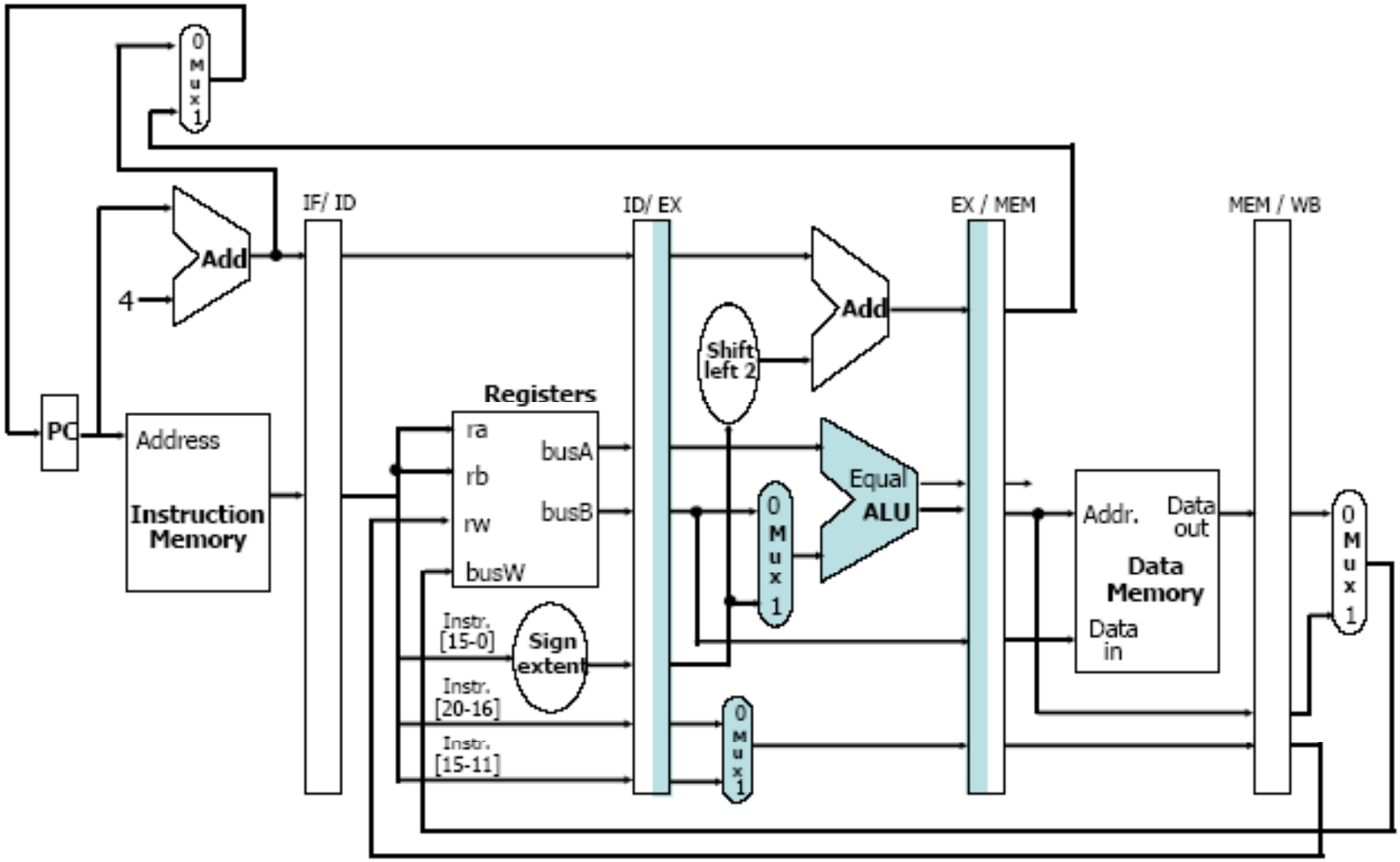
IF:اولین مرحله لوله دستورالعمل بارکردن



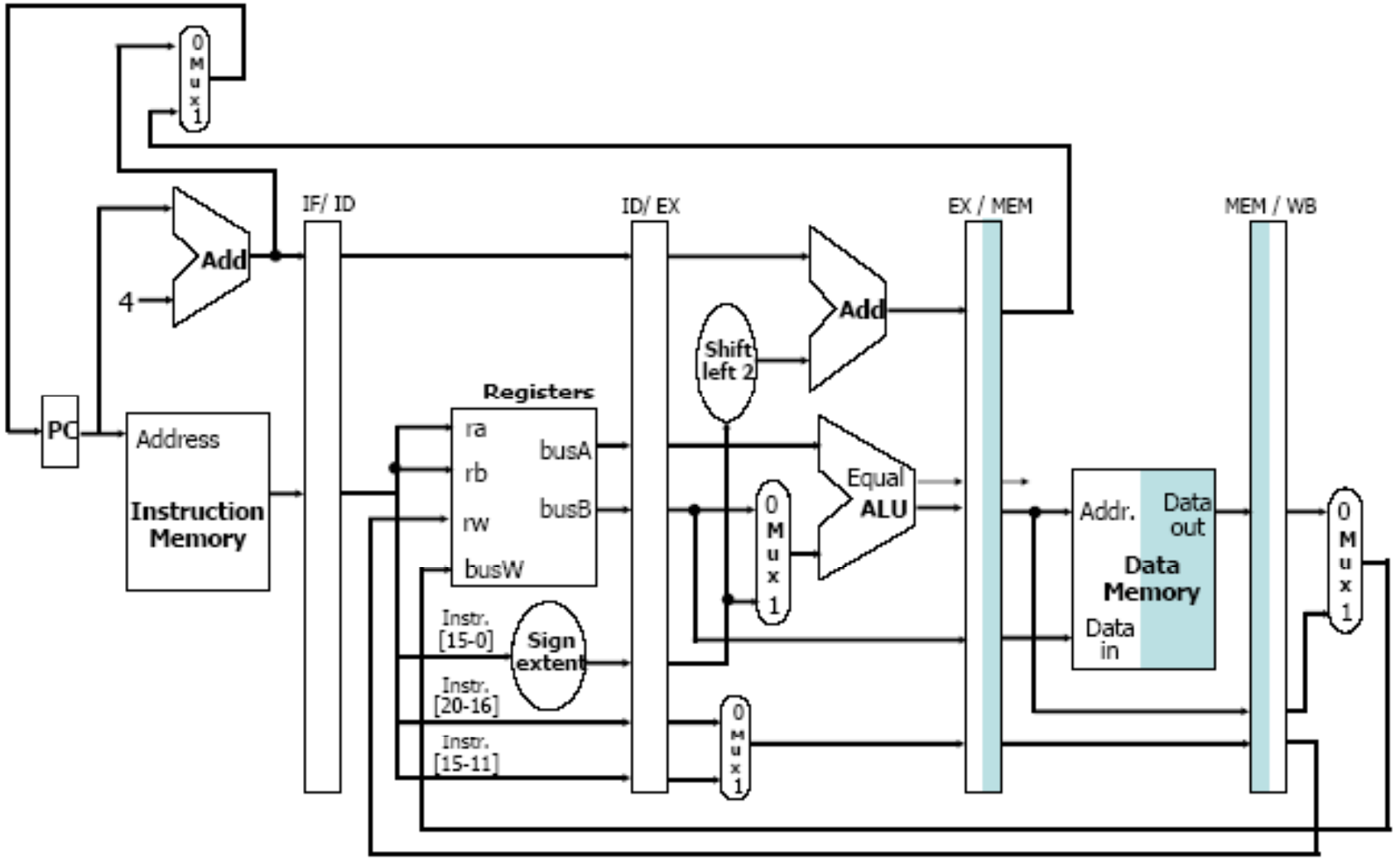
ID:دومین مرحله لوله دستورالعمل بارکردن



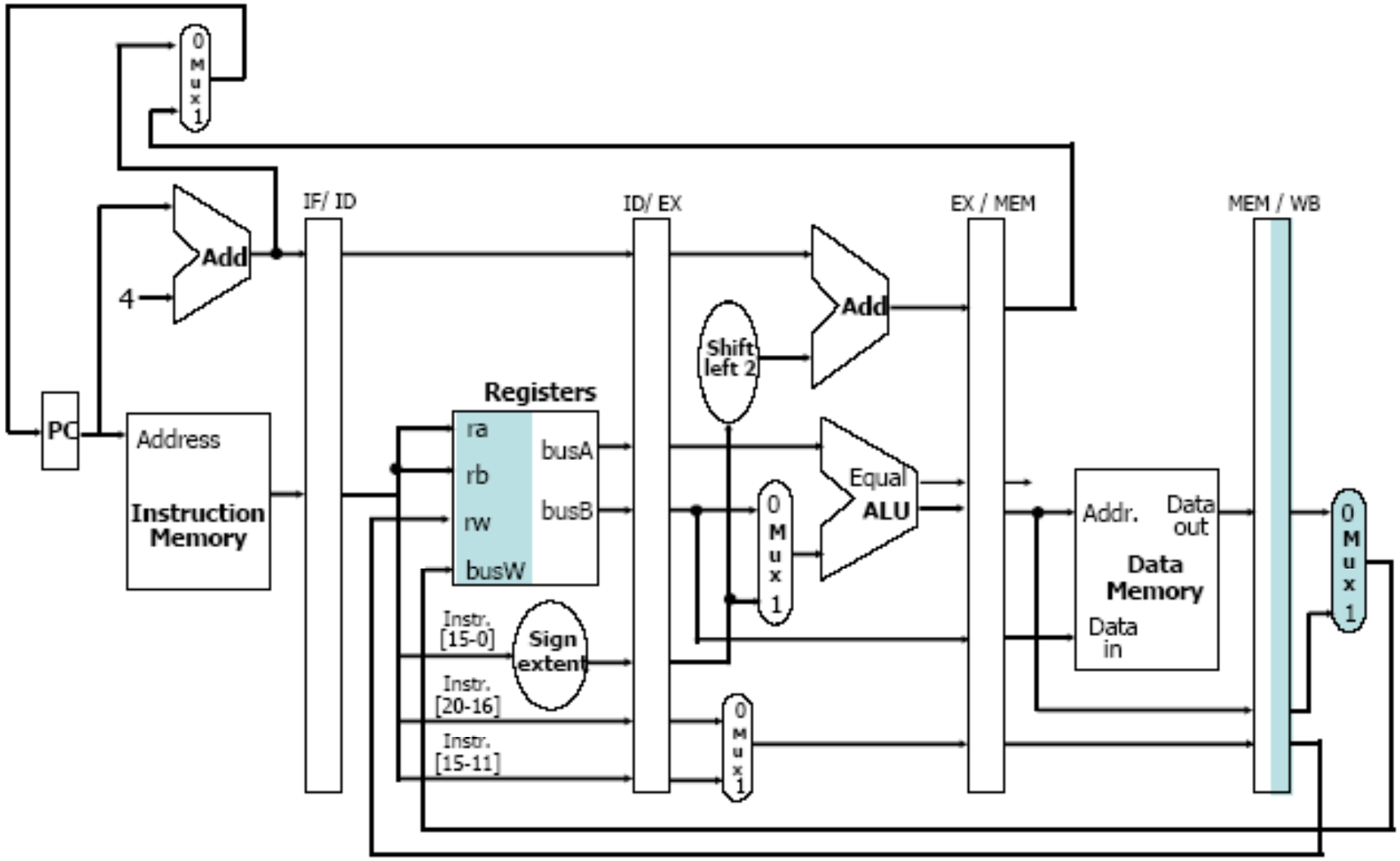
EX:سومین مرحله لوله دستورالعمل بارکردن



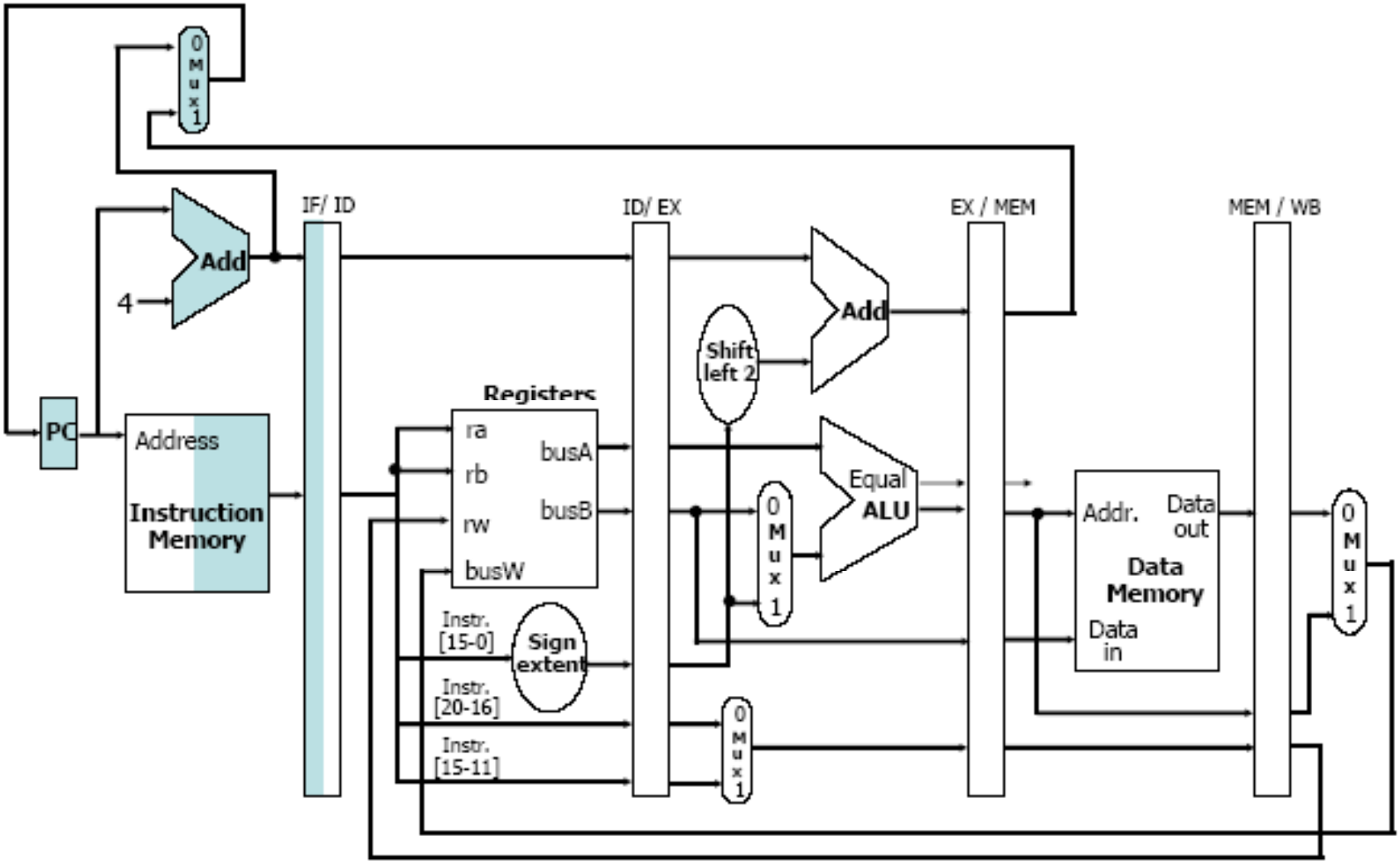
MEM: چهارمین مرحله لوله دستورالعمل بارکردن



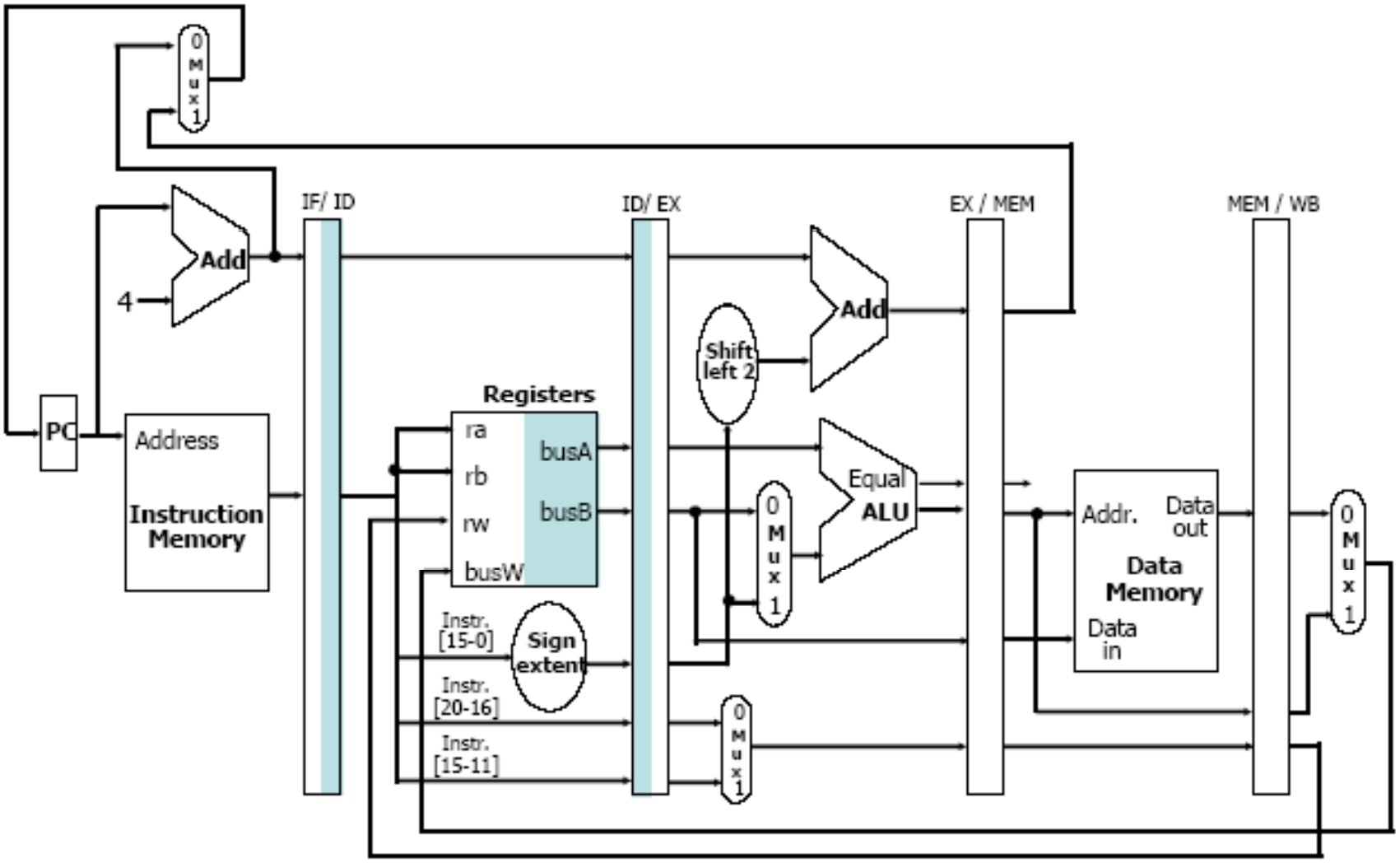
WB:پنجمین مرحله لوله دستورالعمل بارکردن



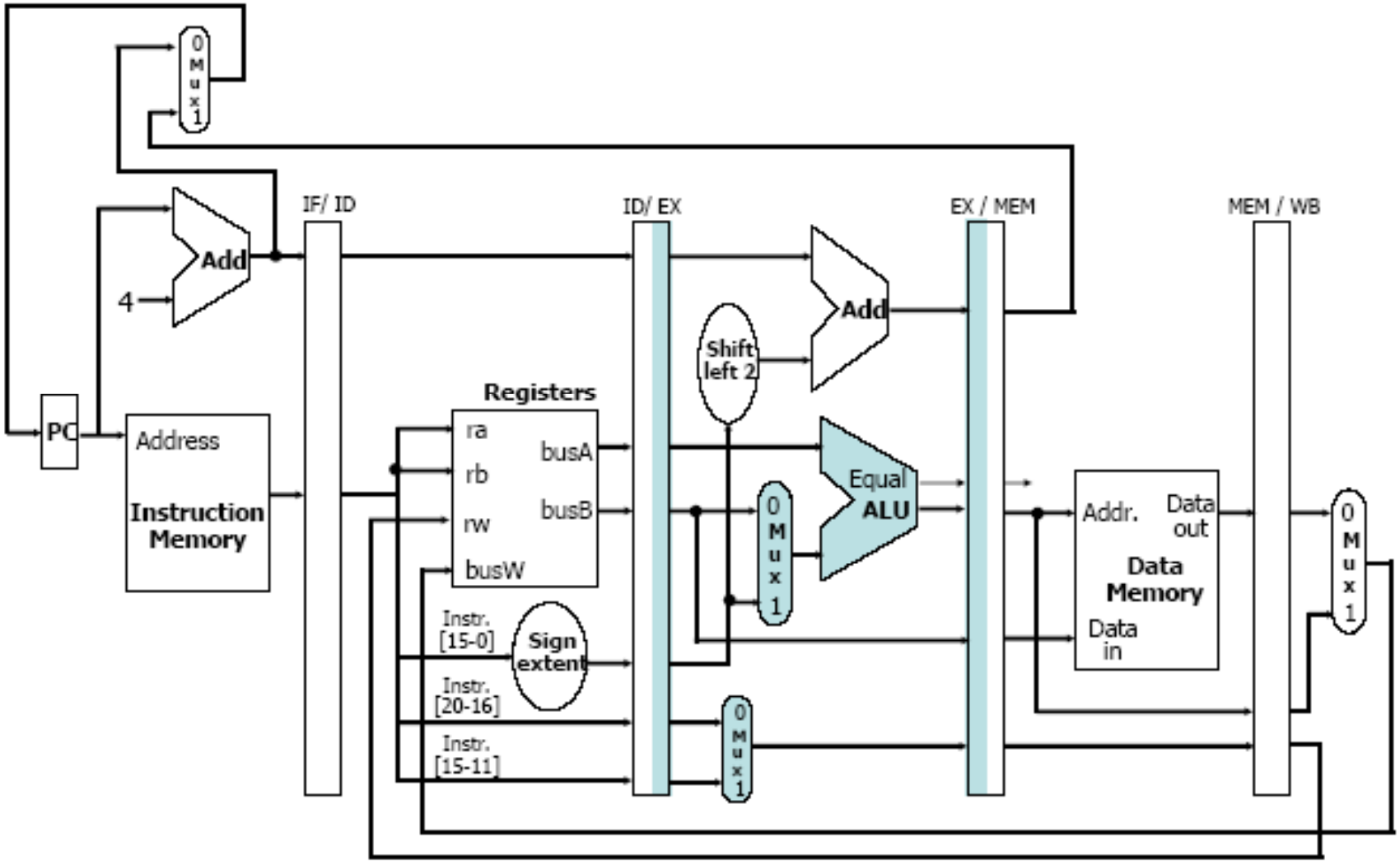
IF:اولین مرحله لوله دستورالعمل R-type



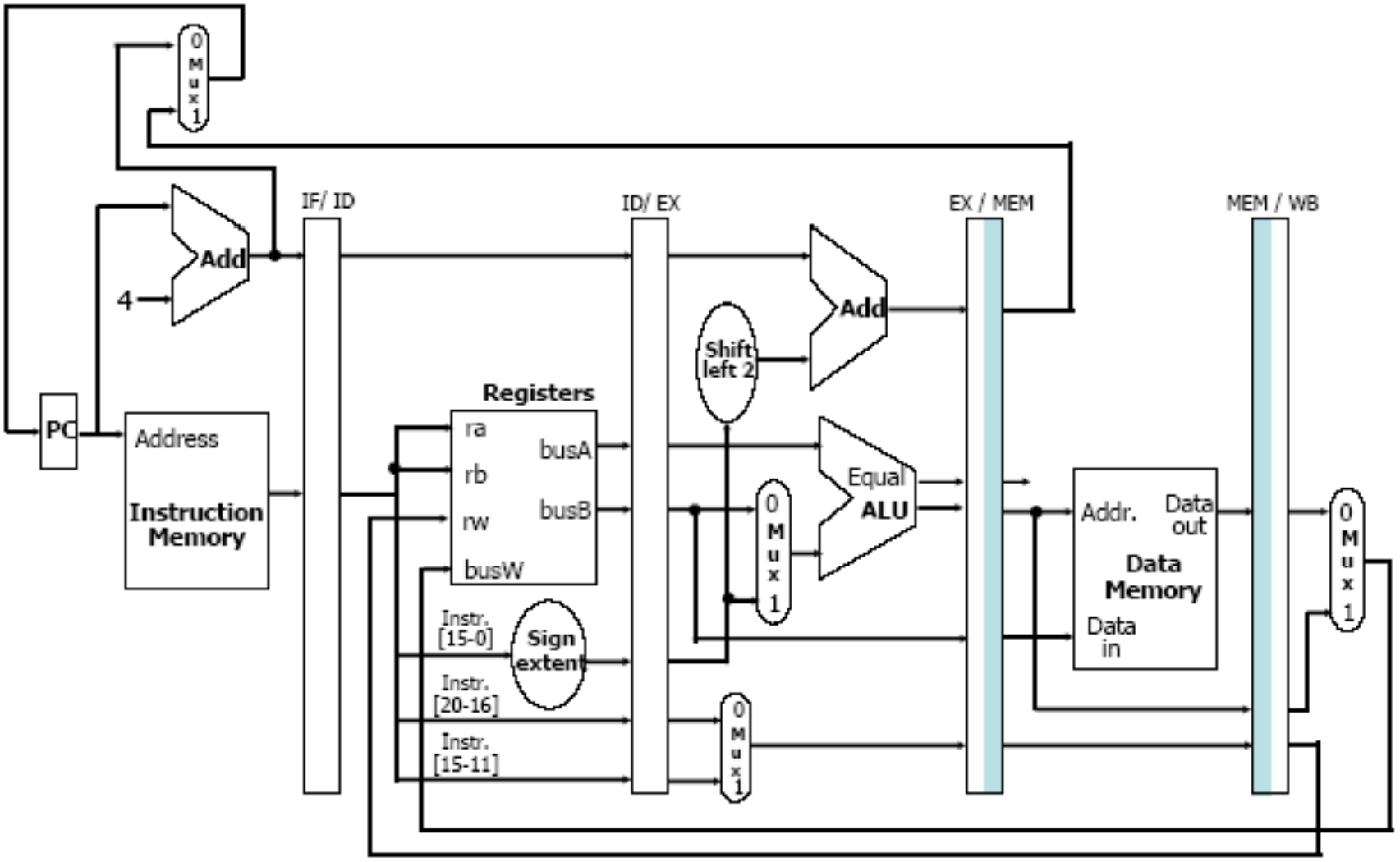
ID:دومین مرحله لوله دستورالعمل R-type



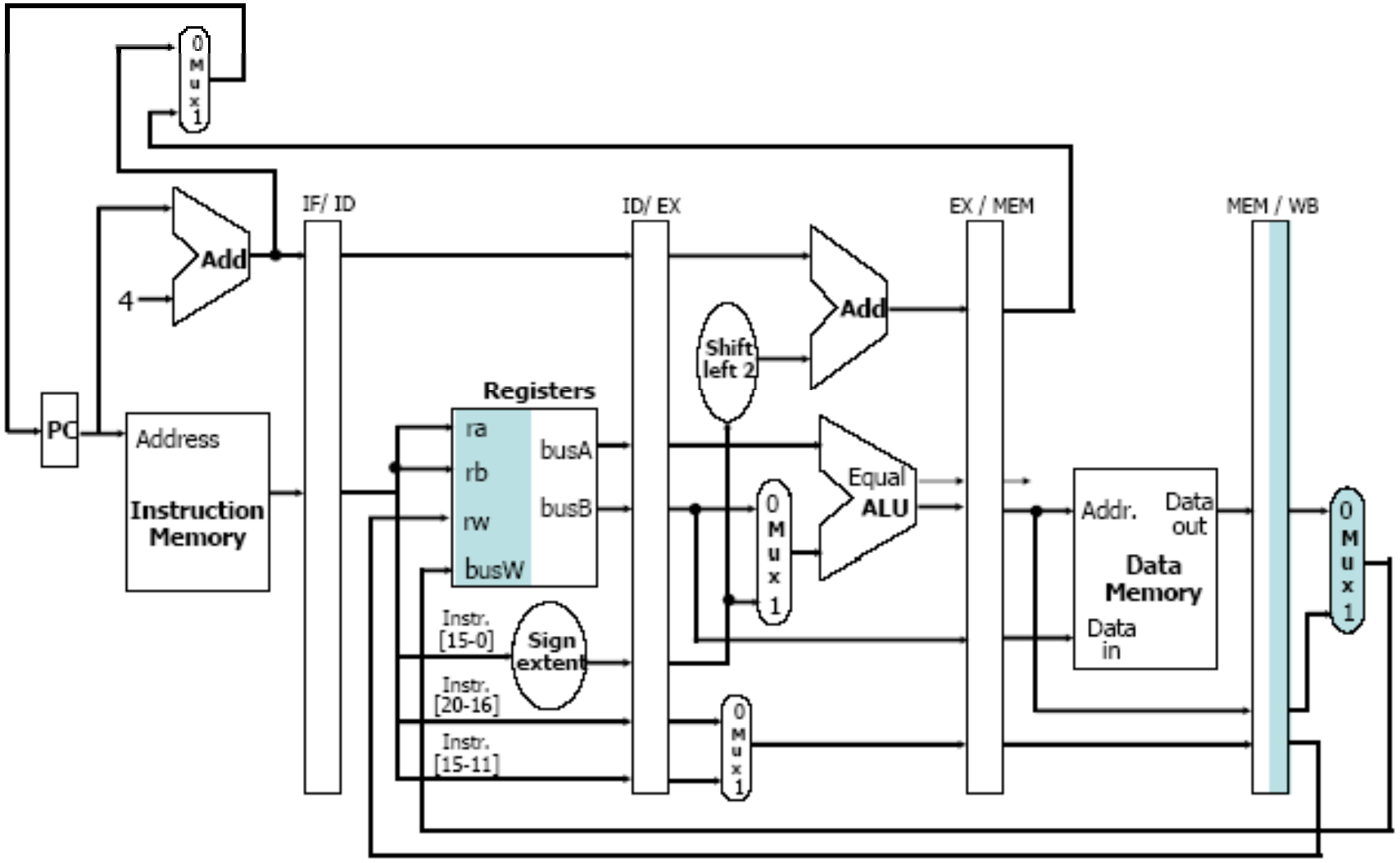
EX:سومین مرحله لوله دستورالعمل R-type



MEM:چهارمین مرحله لوله دستورالعمل R-type



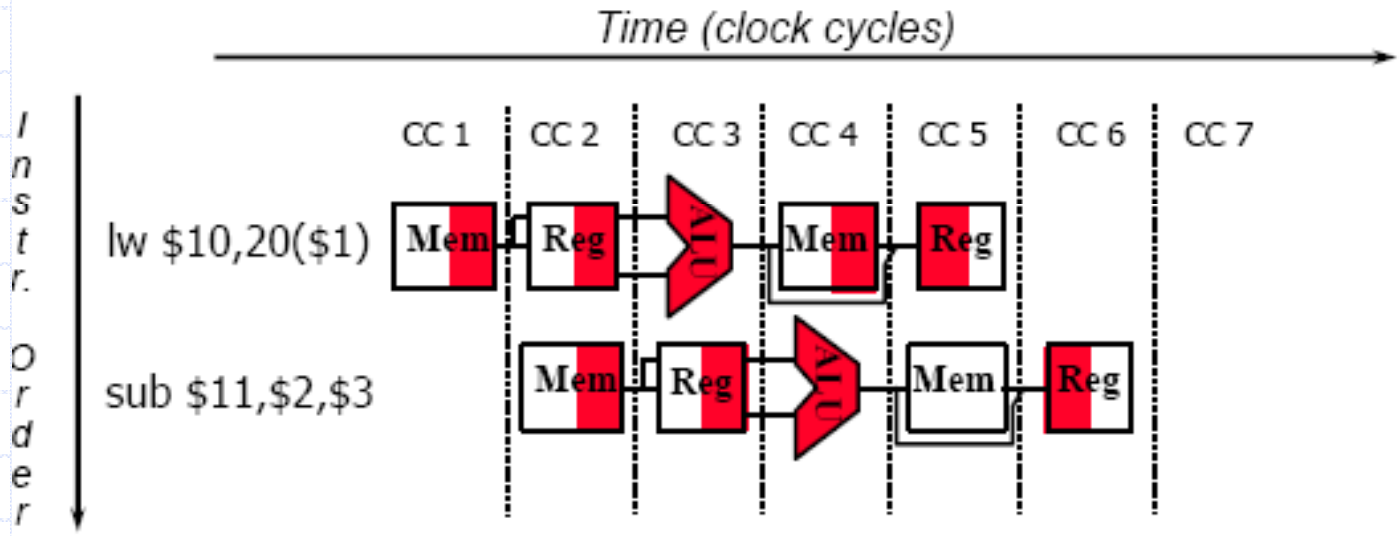
پنجمین مرحله لوله دستورالعمل R-type:WB



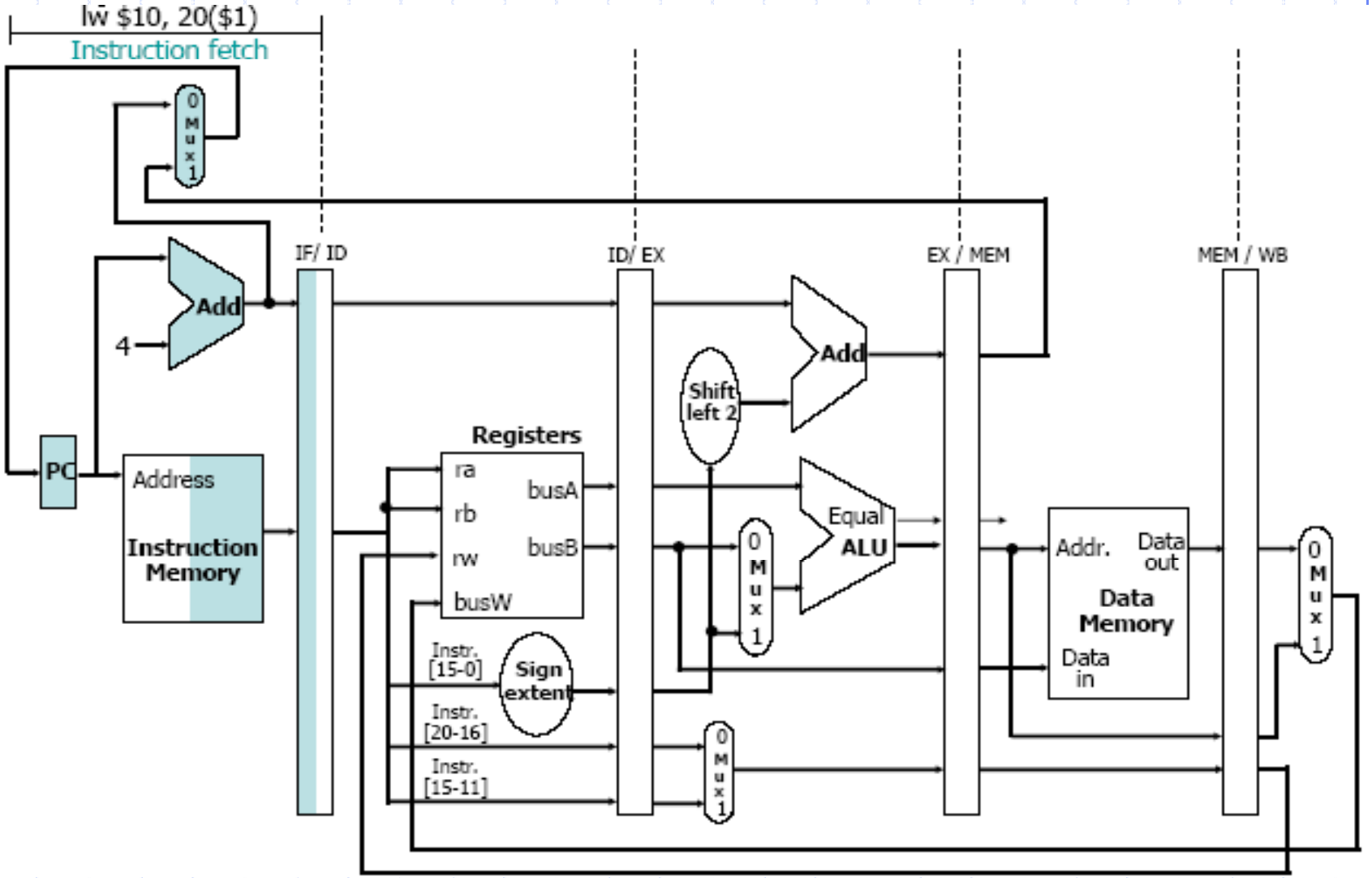
یک مثال برای روشن شدن خط لوله

- از زمانی که بیشتر دستورالعمل ها باهم در یک مسیر داده تک چرخه ای اجرا می شوند،این می تواند برای درک کردن مشکل باشد
- دنبال کردن کد زیر یک آزمون خواهد بود.

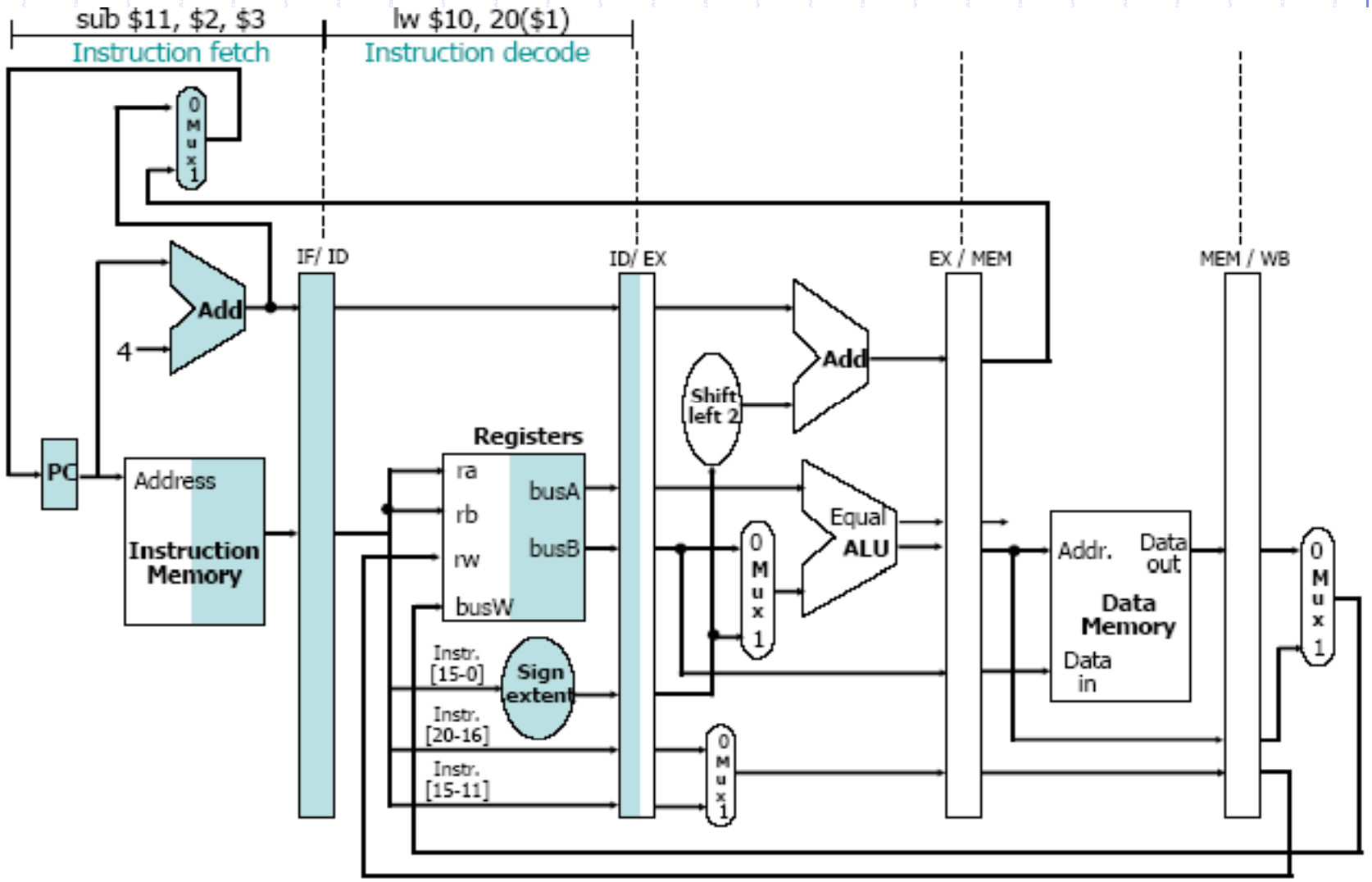
lw $10, 20($1)
sub $11, $2, $3



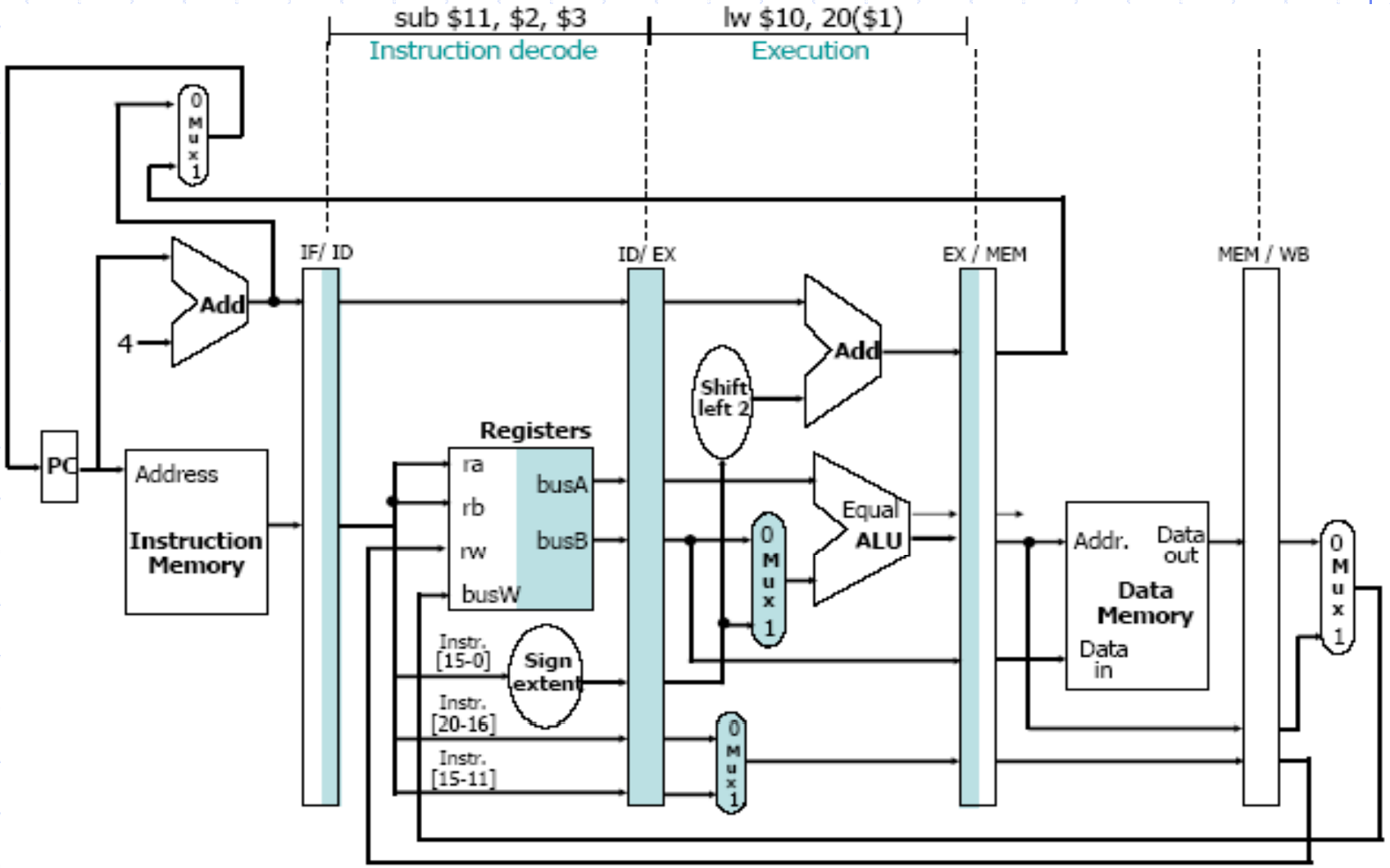
Clock 1


lw $10, 20($1)
Instruction fetch
IF/ ID
ID/ EX
EX / MEM
MEM / WB
M u x
0
1
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Shift left 2
Add
Equal
ALU
Addr.
Data out
Data Memory
Data in
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]

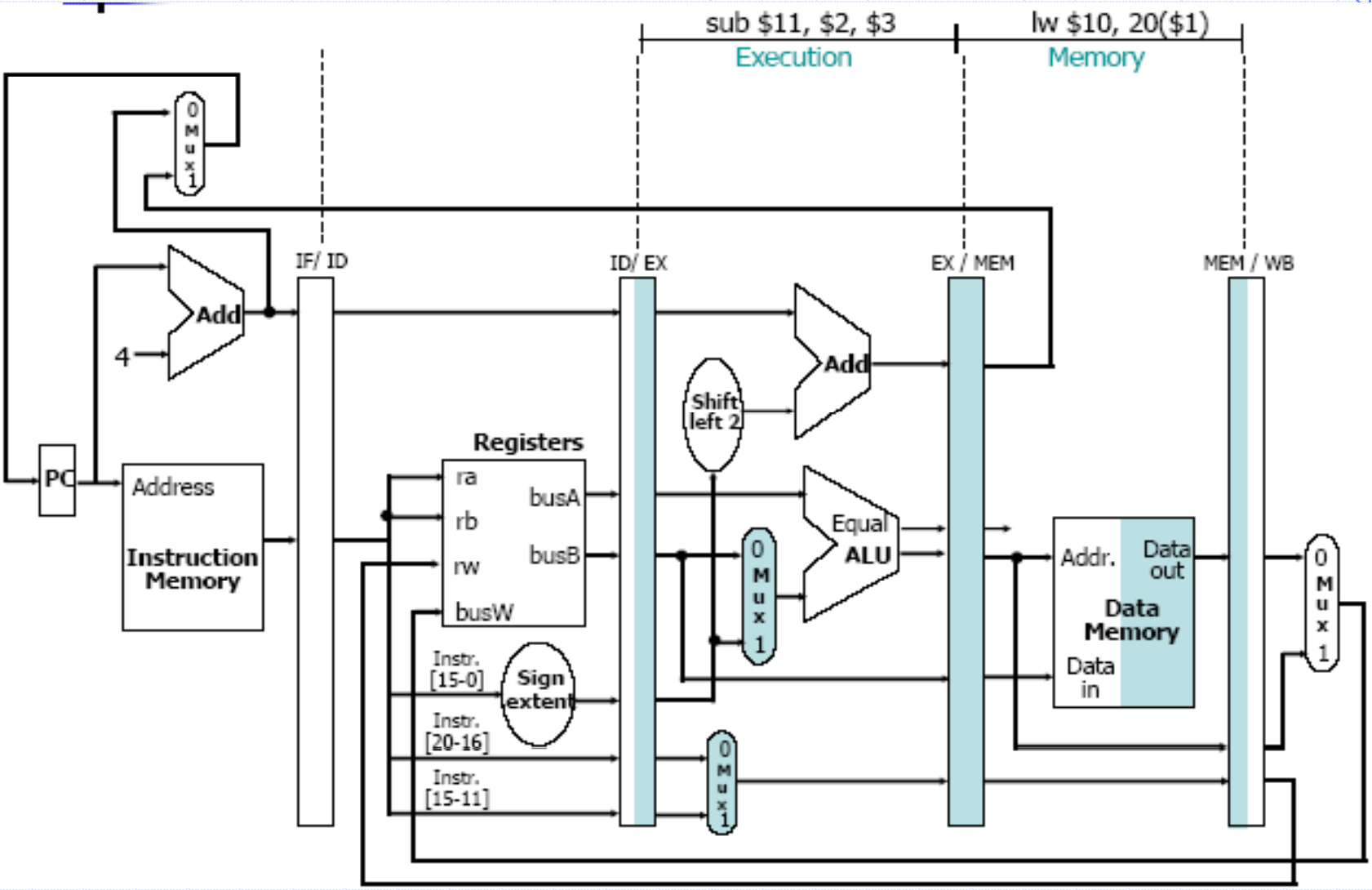
Clock 2


sub $11, $2, $3
lw $10, 20($1)
Instruction fetch
Instruction decode
IF/ ID
ID/ EX
EX / MEM
MEM / WB
M u x
0
1
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Instr. [15-0]
Instr. [20-16]
Instr. [15-11]
Sign extend
Shift left 2
Add
Equal
ALU
Addr.
Data out
Data Memory
Data in

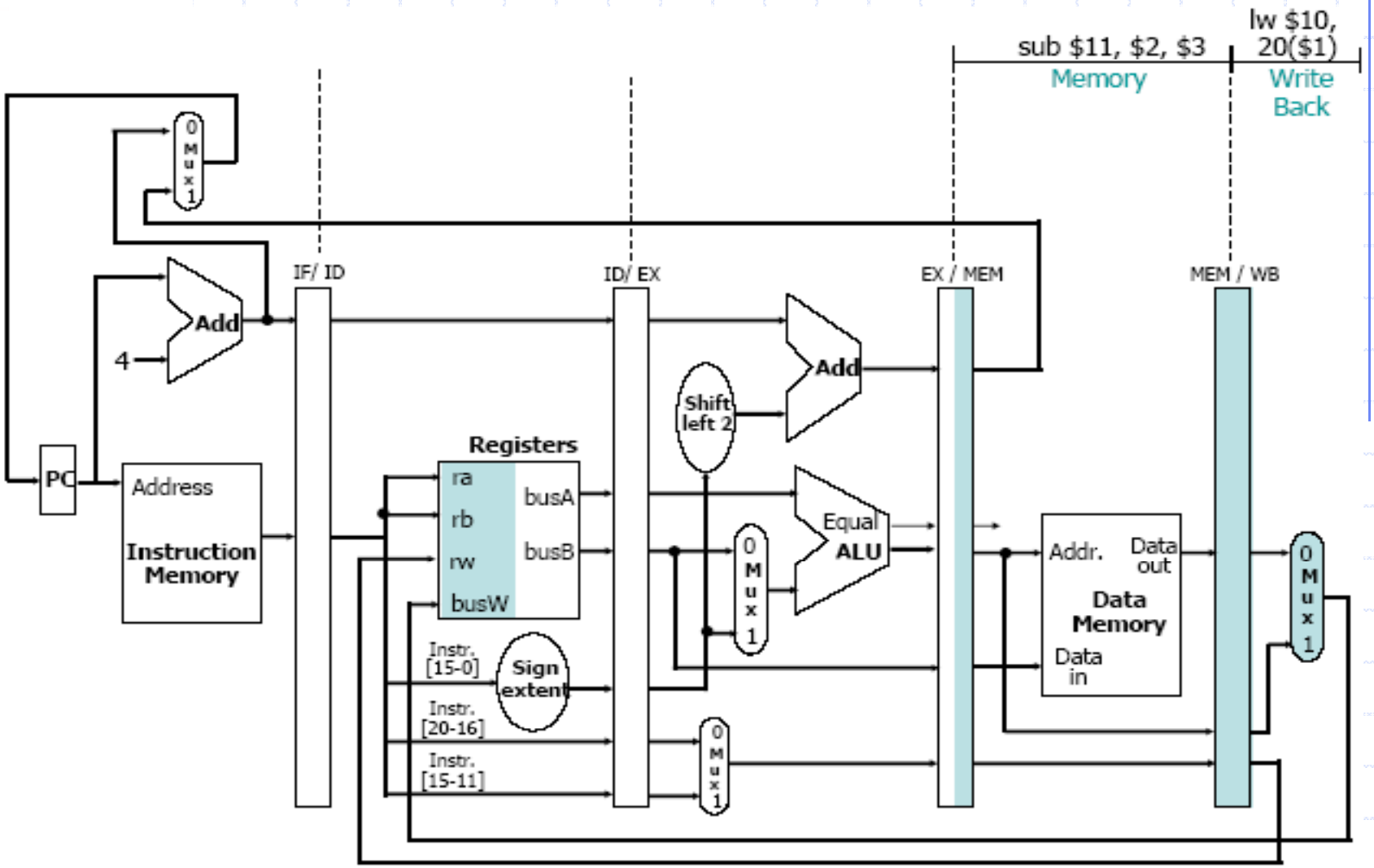
Clock 3


sub $11, $2, $3
Instruction decode
lw $10, 20($1)
Execution
IF/ ID
ID/ EX
EX / MEM
MEM / WB
M u x
0
1
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Shift left 2
Add
Equal
ALU
Addr.
Data out
Data Memory
Data in
Instr. [15-0]
Instr. [20-16]
Instr. [15-11]
Sign extend

Clock 4


sub $11, $2, $3
Execution
lw $10, 20($1)
Memory
IF/ ID
ID/ EX
EX / MEM
MEM / WB
M u x
0
1
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Shift left 2
Add
Equal
ALU
Addr.
Data out
Data Memory
Data in
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]

Clock 5

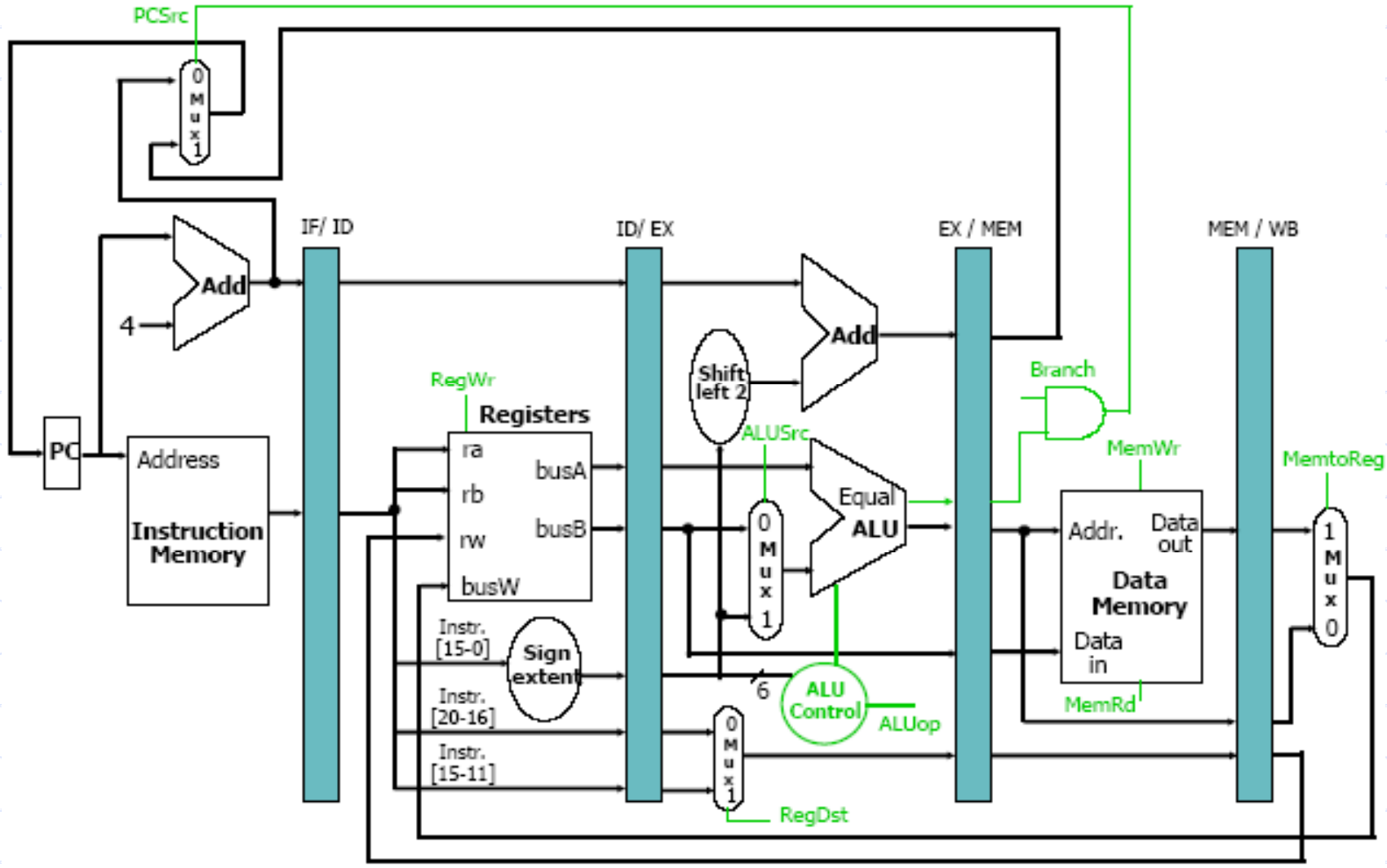

sub $11, $2, $3
lw $10, 20($1)
Memory
Write Back
IF/ ID
ID/ EX
EX / MEM
MEM / WB
M u x
0
1
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Shift left 2
Add
Equal
ALU
Addr.
Data out
Data Memory
Data in
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]

خلاصه:خط لوله ای کردن

•چه چیزی آن را آسان می سازد؟
همه دستورالعمل هایی که طول یکسان دارند
فقطیک تعداد کمی از قالب های دستورالعمل
عملوندهای حافظه تنها در بارکردن و ذخیره سازی ظاهر می شوند
•چه چیزی آن را سخت می سازد؟
هزاردهای ساختاری:فرض کنید ما فقط یک حافظه داریم
هزاردهای کنترلی:ما باید نگران دستورالعمل های انشعاب باشیم
هزاردهای داده ای:یک دستورالعمل وابسته به دستورالعمل های قبلی است
•خط لوله ای کردن یک مفهوم بنیادی است
چندین مرحله استفاده می شود برای جدا کردن مراحل
•پردازشگرهای جدید واقعآ آن را سخت می کنند
استثنا گردانی
سعی کردن برای افزایش کارایی با یک اجرای نادرست و...

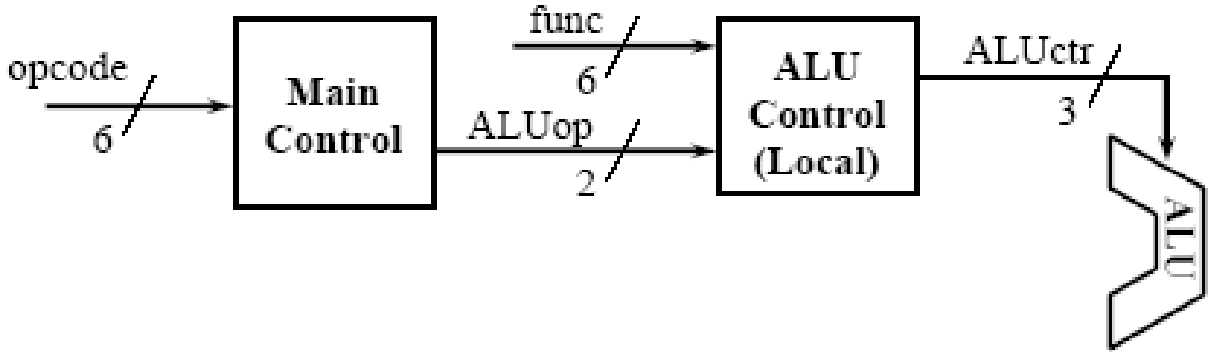
کنترل مسیر داده خط لوله ای شده

مسیر داده خط لوله ای شده با سیگنال های کنترلی



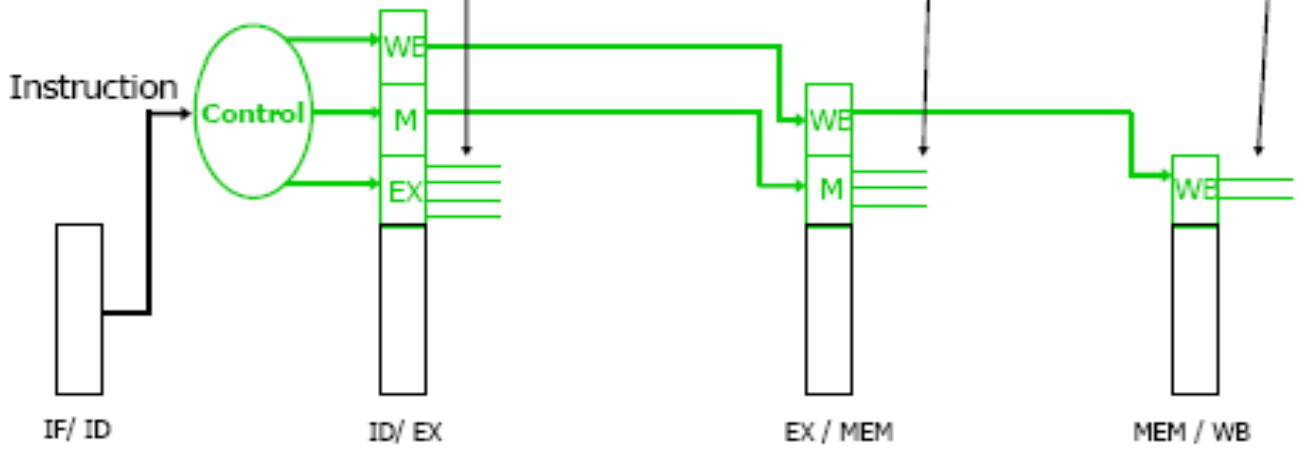
فراخوانی:بیتهای کنترلی ALU

Instruction opcode	ALUop	Instruction operation	Function field	Desired ALU action	ALU control
LW	00	load word	XXXXXX	add	**010**
SW	00	store word	XXXXXX	add	**010**
beq	01	branch equal	XXXXXX	subtract	**110**
R-type	10	add	100000	add	**010**
R-type	10	subtract	100010	subtract	**110**
R-type	10	AND	100100	and	**000**
R-type	10	OR	100101	or	**001**
R-type	10	set on less than	101010	set on less than	**111**

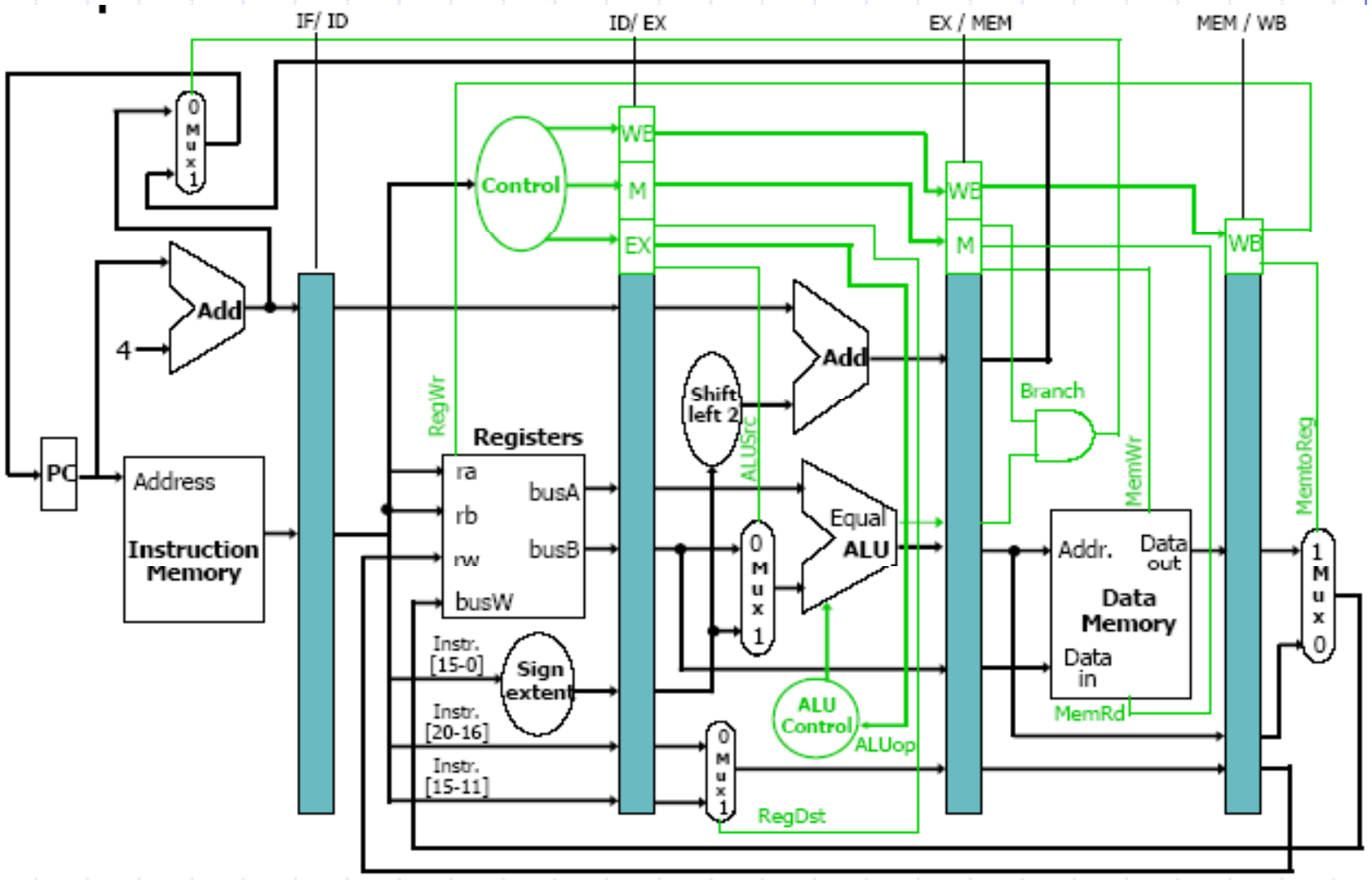


مقادیر خطوط کنترلی برای 3 مرحله آخر خط لوله

Instructions	Execution / Address Calculation stage control lines				Memory Access stage control lines			Write Back stage control lines	
	Reg Dst	ALU Op0	ALU Op1	ALU Src	Branch	Mem Rd	Mem Wr	Reg Wr	Mem to Reg
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
Beq	X	0	1	0	1	0	0	0	X



مسیر داده خط لوله ای شده با سیگنال های کنترلی

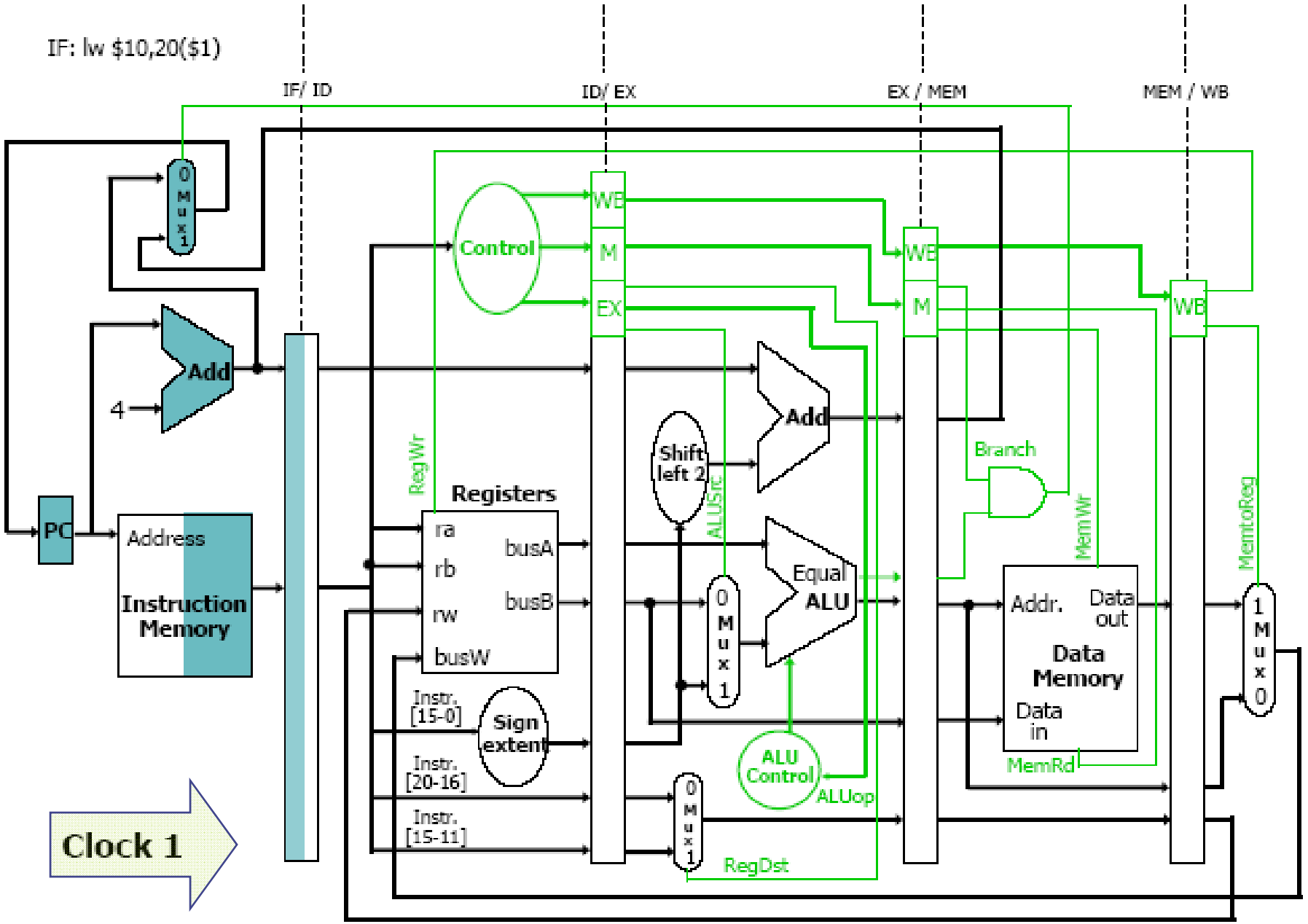


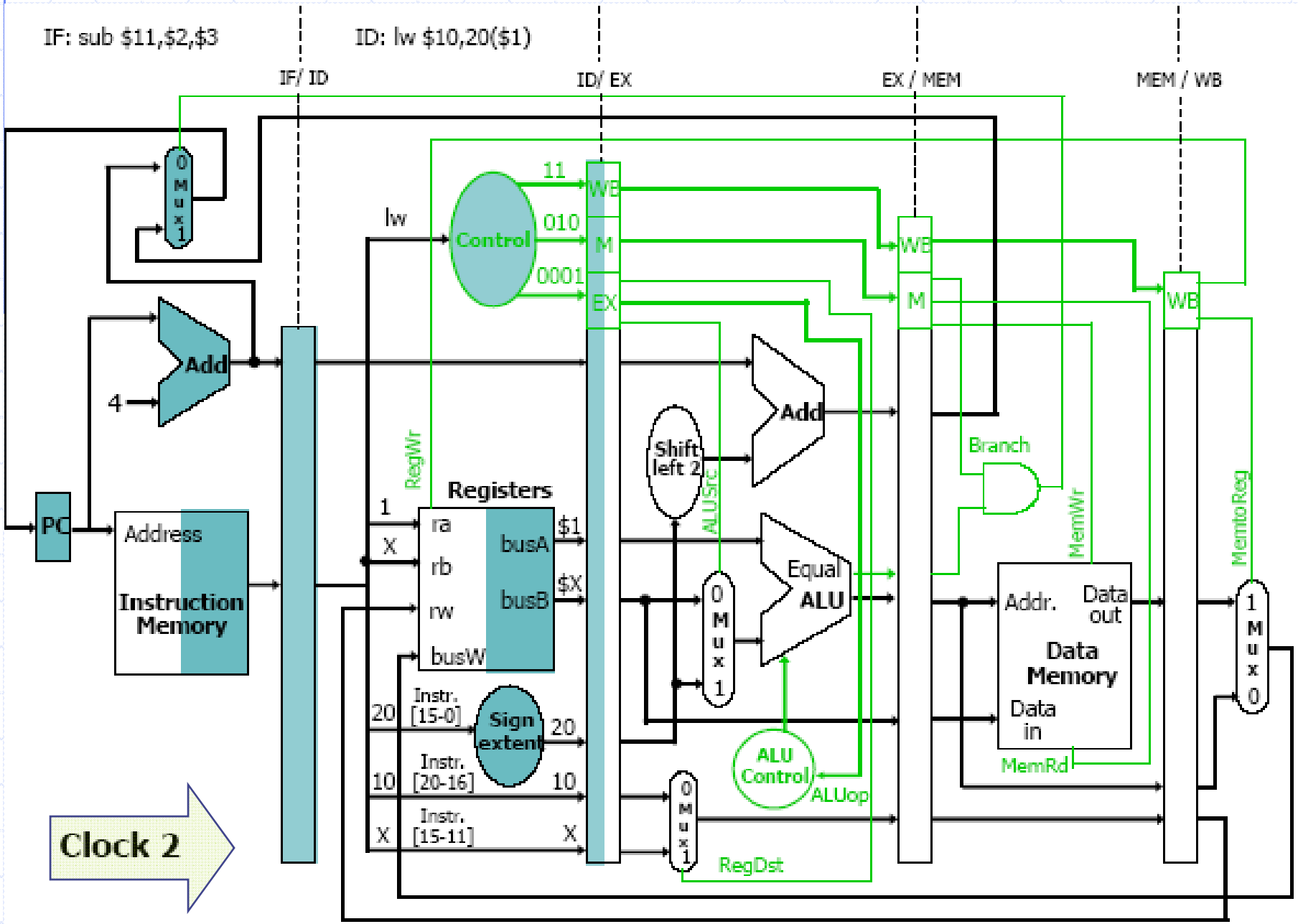
یک مثال برای روشن شدن مسئله کنترل خط لوله

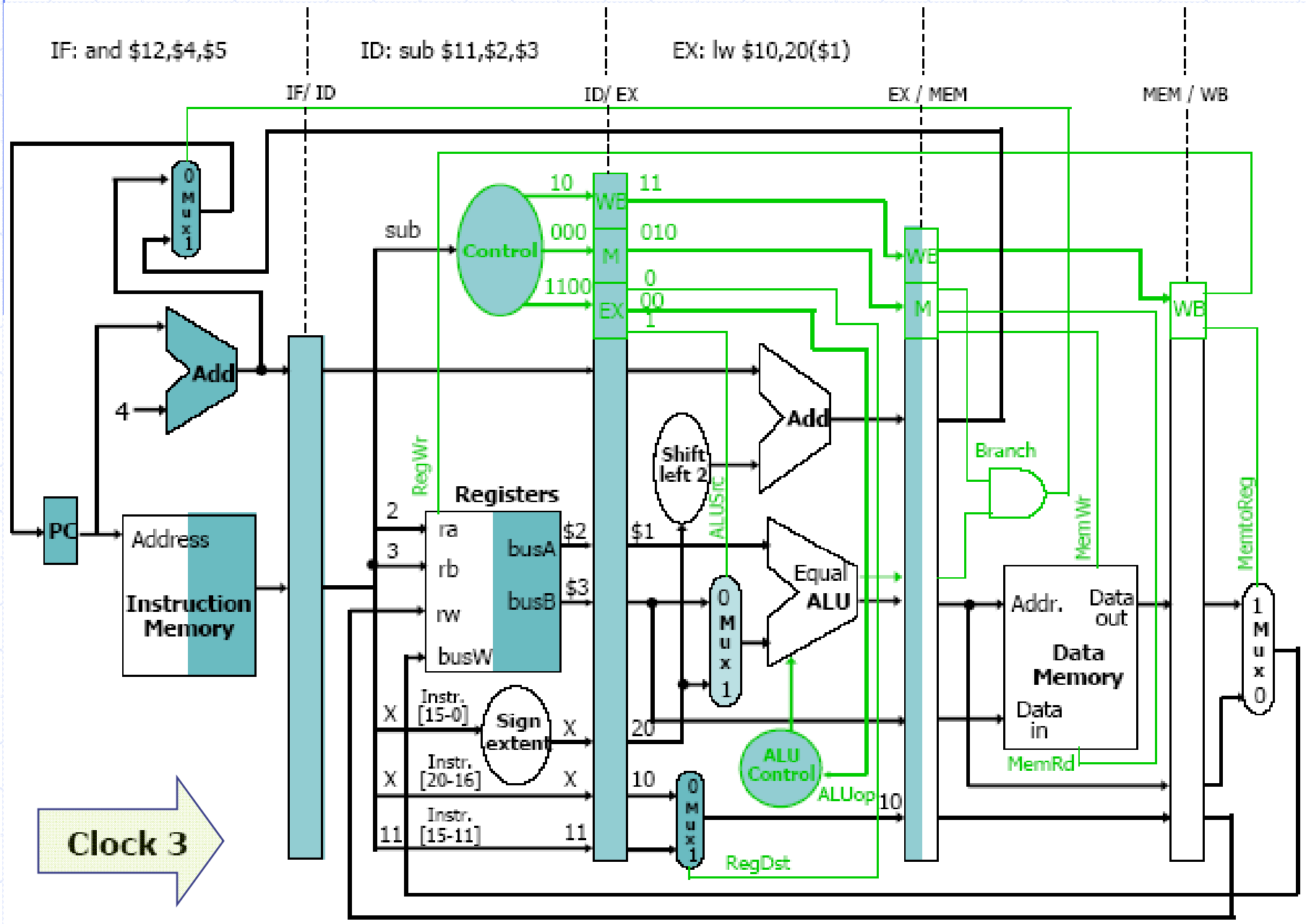
- اجازه بدهید ببینیم چه اتفاقی برای این خط لوله در برنامه زیر می افتد

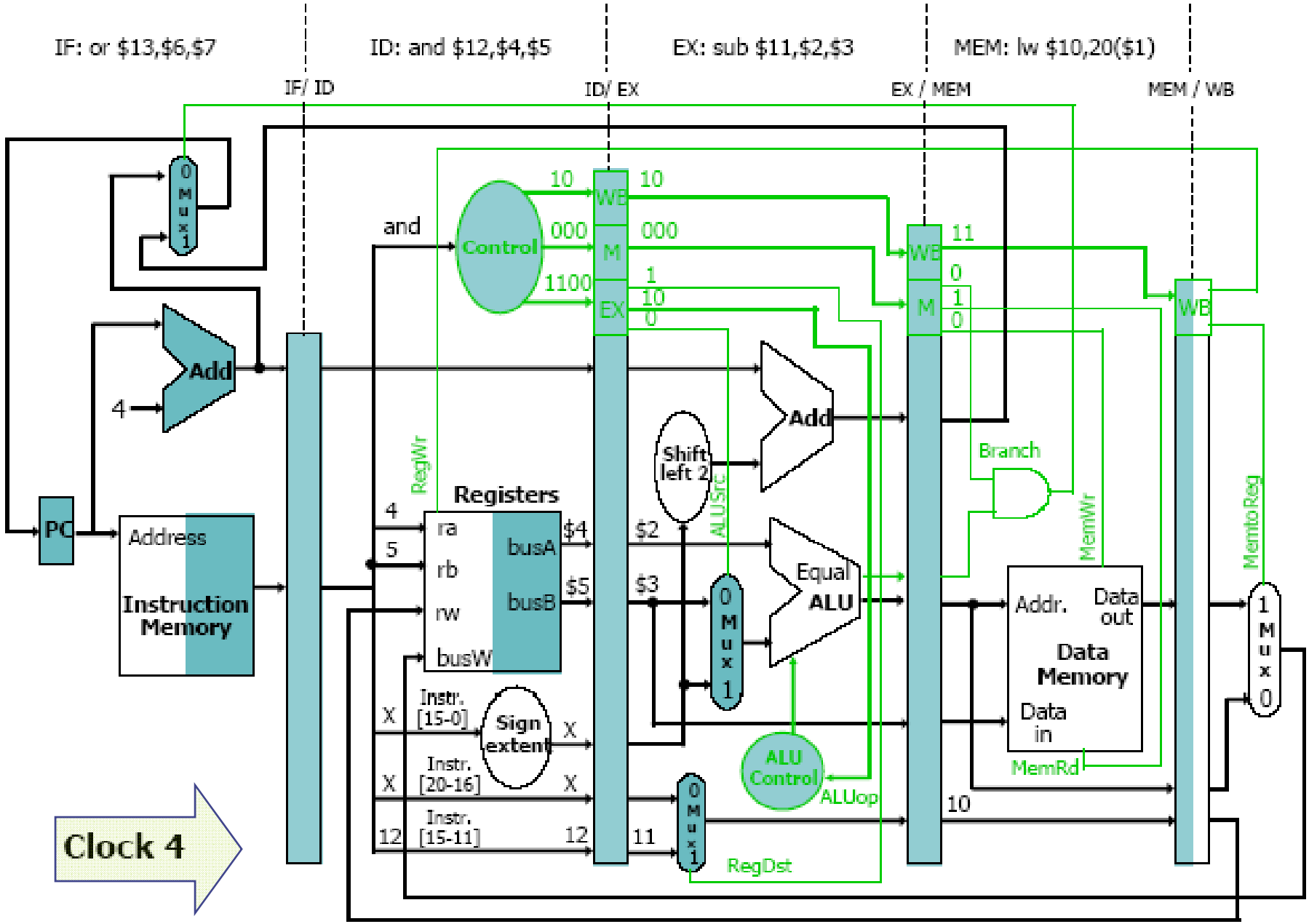
```
lw $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or $13, $6, $7
add $14, $8, $9
```

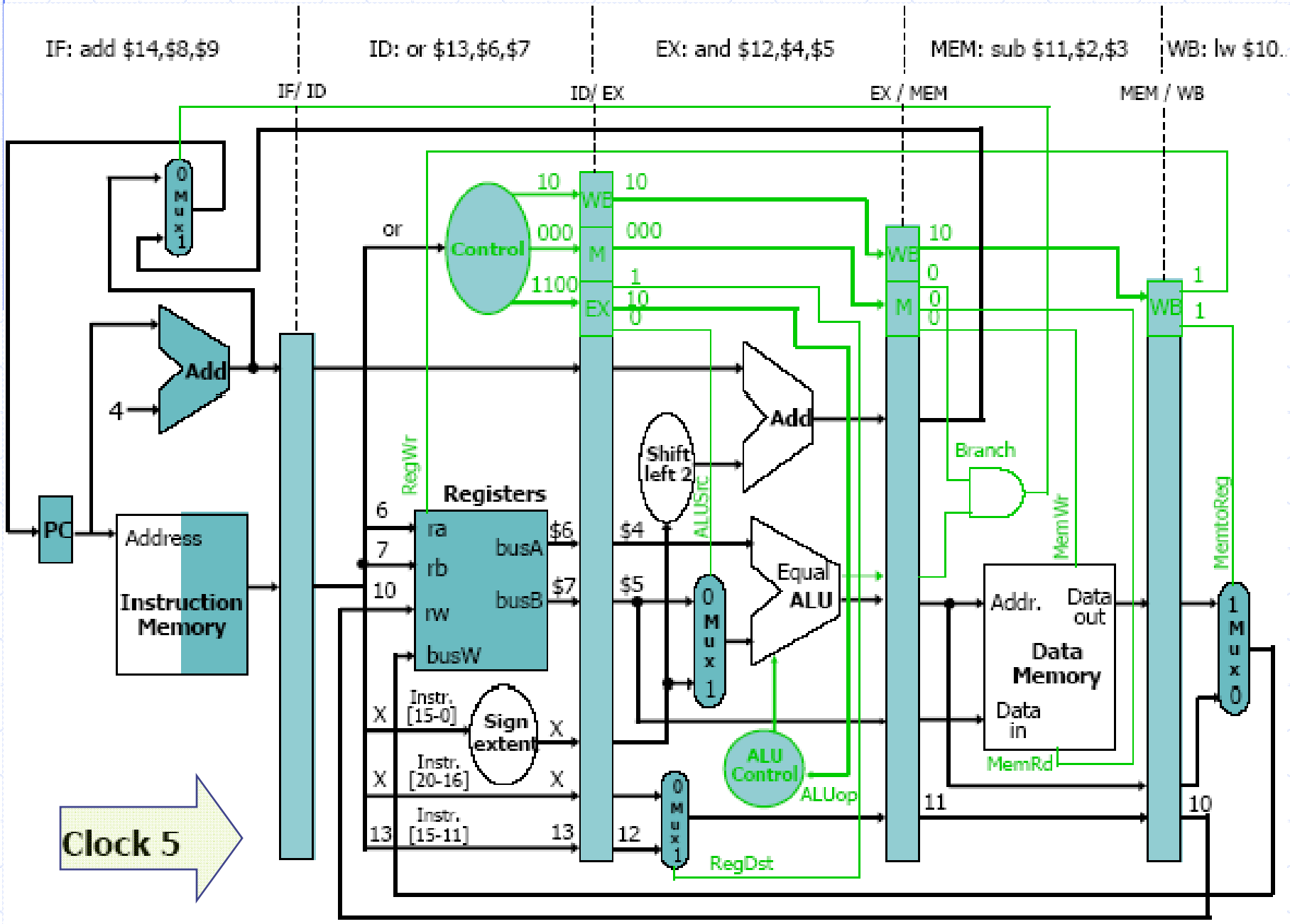
این کد هیچ هزارد داده ای ،ساختاری و کنترلی ندارد


IF: lw $10,20($1)
IF/ ID
ID/ EX
EX / MEM
MEM / WB
0 M u x 1
Add
4
PC
Address
Instruction Memory
Control
WB
M
EX
RegWr
Registers
ra
rb
rw
busW
busA
busB
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]
Shift left 2
ALUSrc
Add
Equal
ALU
0 M u x 1
ALU Control
ALUop
0 M u x 1
RegDst
Branch
MemWr
Addr.
Data out
Data Memory
Data in
MemRd
MemtoReg
1 M u x 0
Clock 1


IF: sub $11,$2,$3
ID: lw $10,20($1)
IF/ ID
ID/ EX
EX / MEM
MEM / WB
0 M u x 1
Add
4
PC
Address
Instruction Memory
lw
Control
11
010
0001
WB
M
EX
RegWr
Registers
1
X
ra
rb
rw
busW
busA
busB
$1
$X
Instr. [15-0]
Instr. [20-16]
Instr. [15-11]
20
10
X
Sign extend
Shift left 2
ALUSrc
Add
Equal
ALU
0 M u x 1
ALU Control
ALUop
RegDst
Branch
MemWr
MemRd
MemtoReg
Addr.
Data out
Data Memory
Data in
1 M u x 0
Clock 2


IF: and $12,$4,$5
ID: sub $11,$2,$3
EX: lw $10,20($1)
IF/ ID
ID/ EX
EX / MEM
MEM / WB
0 M u x 1
Add
4
PC
Address
Instruction Memory
sub
Control
10
000
1100
WB
M
EX
11
010
0
00
1
WB
M
WB
Add
Shift left 2
RegWr
Registers
2
3
ra
rb
rw
busW
busA
busB
$2
$3
$1
ALUSrc
Branch
MemWr
MemtoReg
Equal
ALU
0 M u x 1
Addr.
Data out
Data Memory
Data in
MemRd
1 M u x 0
X
Instr. [15-0]
Sign extent
X
20
X
Instr. [20-16]
X
10
ALU Control
ALUop
10
11
Instr. [15-11]
11
0 M u x 1
RegDst
Clock 3


IF: or $13,$6,$7
ID: and $12,$4,$5
EX: sub $11,$2,$3
MEM: lw $10,20($1)
IF/ ID
ID/ EX
EX / MEM
MEM / WB
and
Control
10
000
1100
WB
M
EX
10
000
1
10
0
11
0
1
0
WB
Add
4
PC
Address
Instruction Memory
RegWr
Registers
ra
rb
rw
busW
busA
busB
4
5
$4
$5
$2
$3
Shift left 2
ALUSrc
Add
Branch
Equal
ALU
MemWr
MemtoReg
Addr.
Data out
Data Memory
Data in
MemRd
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]
X
X
X
X
12
12
11
ALU Control
ALUop
10
RegDst
Clock 4


IF: add $14,$8,$9
ID: or $13,$6,$7
EX: and $12,$4,$5
MEM: sub $11,$2,$3
WB: lw $10..
IF/ ID
ID/ EX
EX / MEM
MEM / WB
Clock 5
PC
Address
Instruction Memory
Add
4
Registers
ra
rb
rw
busW
busA
busB
RegWr
6
7
10
$6
$7
Control
or
10
000
1100
WB
M
EX
Shift left 2
ALUSrc
$4
$5
Equal
ALU
Branch
MemWr
Addr.
Data out
Data Memory
Data in
MemRd
MemtoReg
ALU Control
ALUop
Instr. [15-0]
Sign extend
Instr. [20-16]
Instr. [15-11]
X
13
12
11
RegDst
1
0

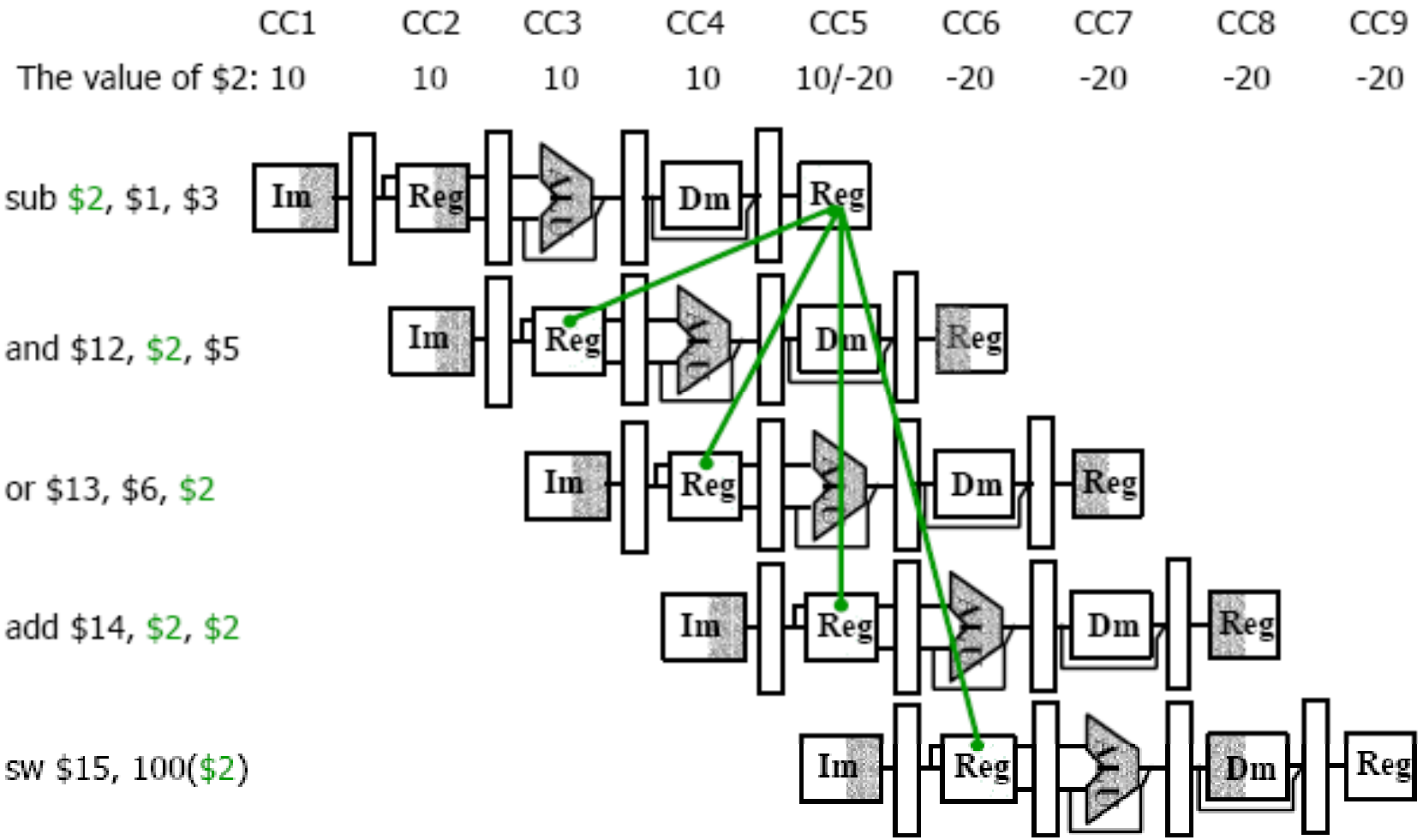
هزاردهای داده ای

- مثال قبل به ما نشان می دهد که دستورالعمل های مستقل که از نتایج آنها در دستورات قبل استفاده نمی شود چطور اجرا شده اند.
- این یک نمونه با برنامه های واقعی نیست.
- اجازه دهید کدهای مرتب زیر زا دنبال کنیم.

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- 4 دستور آخر همگی به ثبات $2وابسته اند که در دستور اول تولید می شود.
- فرض کنید ثبات $2مقدار 10 را قبل از عمل تفریق دارد و مقدار 20 را بعد از آن دارد.

هزاردهای داده ای



هزاردهای داده ای

راه حل ساده : کامپایلر هیچ دستورالعملی را بین دستورالعمل های sub-and وارد نمی کند

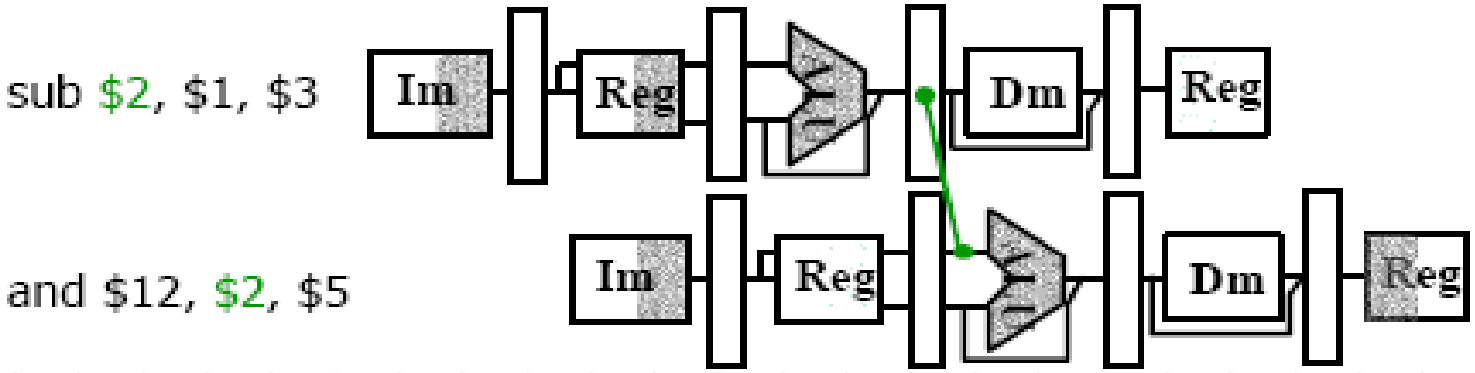
- هیچ یک از این دو دستورالعمل نه داده ای را تعریف می کنند و نه نتیجه ای را می نویسند.

```
sub $2, $1, $3
nop
nop
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

- **نتیجه :** این کار انجام می شود اما 2 چرخه ساعت به هدر می رود (کارایی کاهش می یابد)

کشف هزاردهای داده ای و ارسال(forwading)

- این ممکن است که هزارد داده ای کشف شود و سپس مقدار مناسب را برای حل هزارد ارسال (Forwarding)کنیم
- زمانی که یک دستورالعمل سعی می کند یک ثبات را در مرحلهEX بخواند دستورالعمل زودتر دیگری قصد دارد در مرحله WBبنویسد.
- این یک نمونه هست بین دستورالعمل های sub-andزیر:

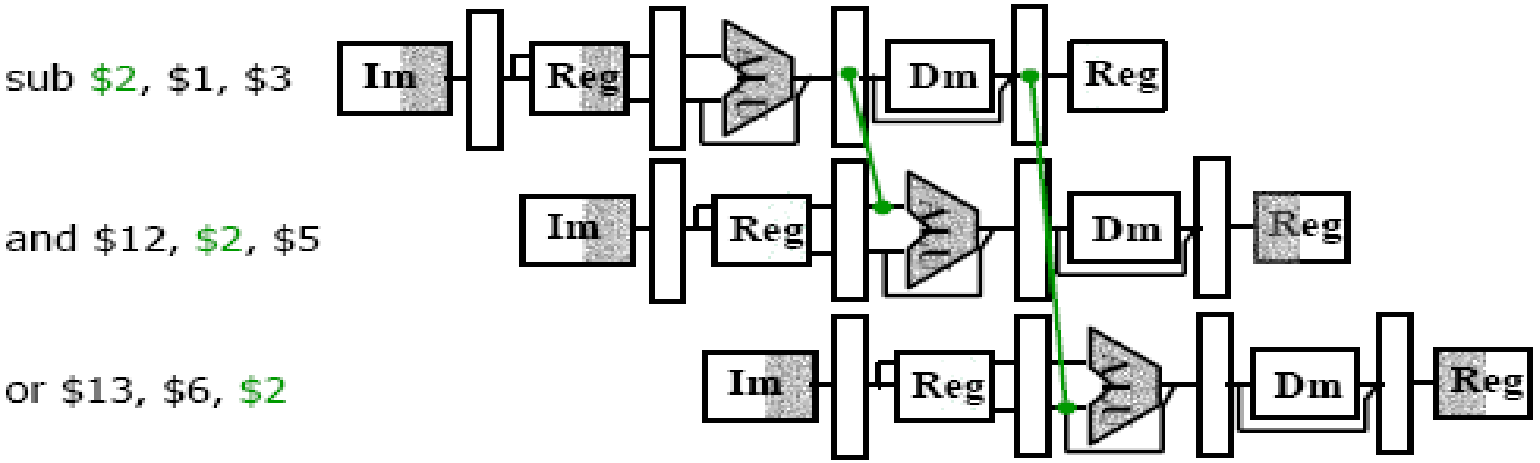


•این هزارد می تواند کشف شود به وسیله یک چک کردن ساده

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2

کشف هزاردهای داده ای و ارسال(forwading)

- هزارد دیگر است بین دستورالعمل های SUB_OR



- این هزارد می تواند کشف شود به وسیله یک چک کردن ساده

MEM/WB.RegisterRd = ID/EX.RegisterRt = $2

- هزارد داده ای بین دستورالعمل های sub-add وsub-sw وجود ندارد

خلاصه ای از وضعیت های هزارد داده ای

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

This actually refers to destination field of an instruction. It is **rd** filed in **R-type** instructions and **rt** field in **I-type** instructions. Mux in the **EX** stage chooses the correct one, therefore, **EX/MEM** and **MEM/WB** pipeline registers store this information as a **rd** filed (EX/MEM.Register**Rd** and MEM/WB.Register**Rd**).

- Since some of the instructions (i.e. `sw`, `beq`) do not write to register file, the above policy is inaccurate. Consider the following code sequence:

```
sw  $1, 100($5)
add $3, $1, $2
```

EX/MEM.RegisterRd = ID/EX.RegisterRs

$1 = 100+$5

- This problem can be solved simply by checking RegWr signal.

خلاصه ای از وضعیت های هزارد داده ای

چه اتفاقی می افتد اگر0$به عنوان ثبات مقصد استفاده شود؟

- هیچ مقدار صفری ارسال نشود
- بنابراین کشف هزارد باید به صورت زیر دنبال شود.

EX hazard:

if (EX/MEM.RegWr
and (EX/MEM.RegisterRd = 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA=10
if (EX/MEM.RegWr
and (EX/MEM.RegisterRd = 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB=10

MEM hazard:

if (MEM/WB.RegWr
and (MEM/WB.RegisterRd = 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA=10
if (MEM/WB.RegWr
and (MEM/WB.RegisterRd = 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB=10

خلاصه ای از وضعیت های هزارد داده ای

- ملاحظه کنید ترتیب زیر را :

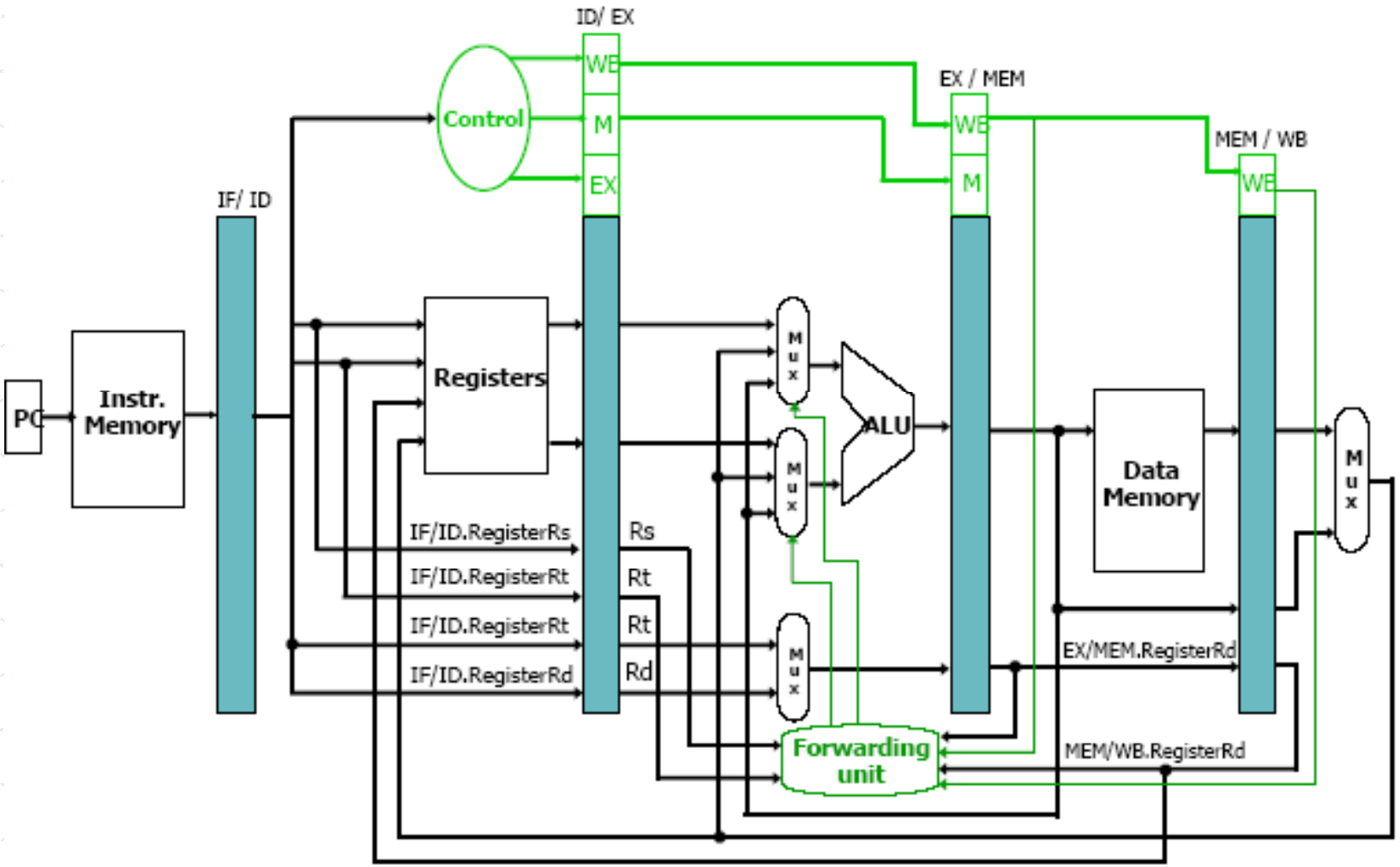
```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4
. . .
```

- در این نمونه ،نتیجه از مرحله MEM ارسال شده برای اینکه نتیجه در مرحله MEM نتیجه اخیر خیلی است نسبت به نتیجه در مرحله WB.

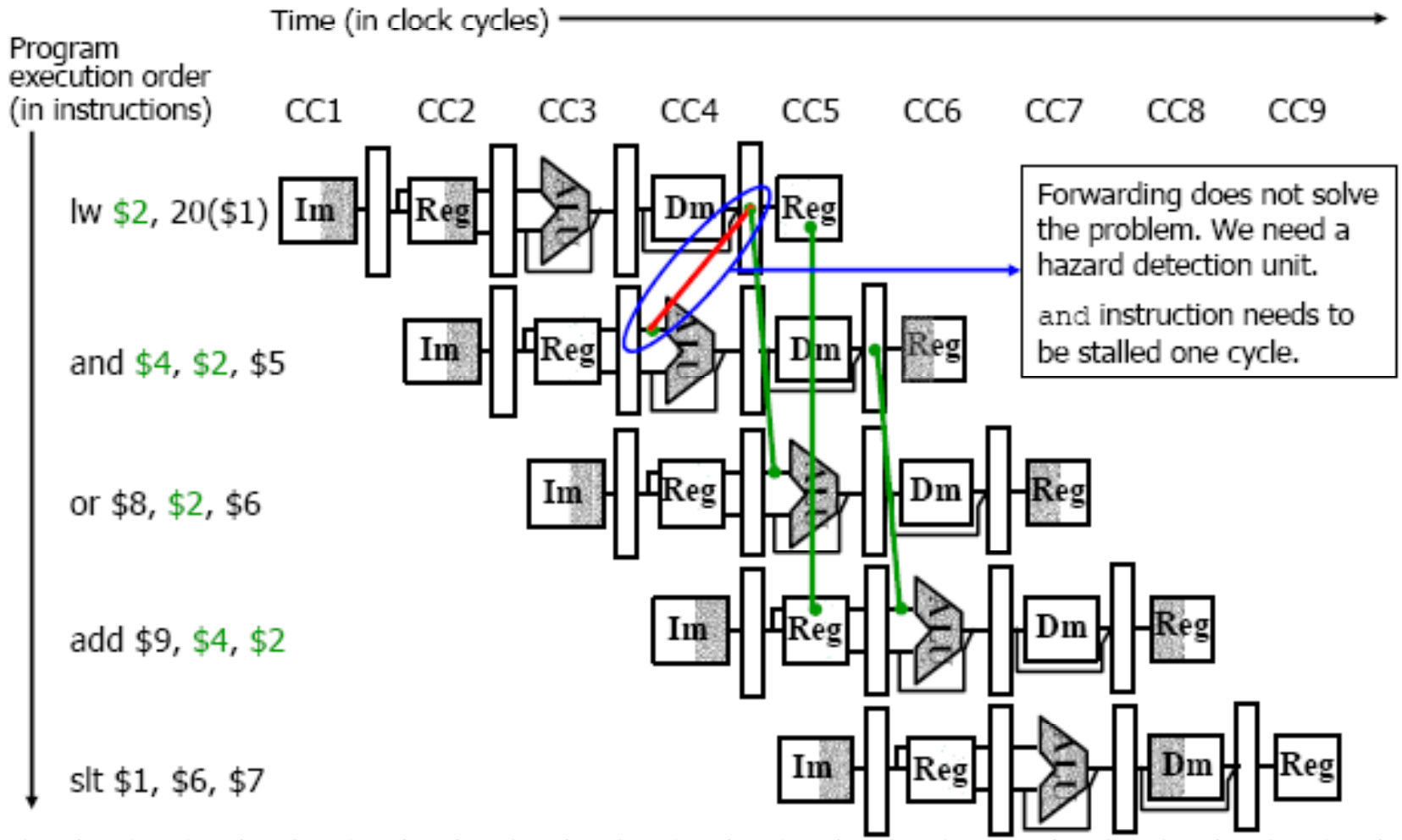
MEM hazard:

```
if (MEM/WB.RegWr
and (MEM/WB.RegisterRd = 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA=10
if (MEM/WB.RegWr
and (MEM/WB.RegisterRd = 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB=10
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)
```

مسیر داده خط لوله ای شده با ارسال(Forwarding)



هزاردهای داده ای و توقف



کشف هزارد

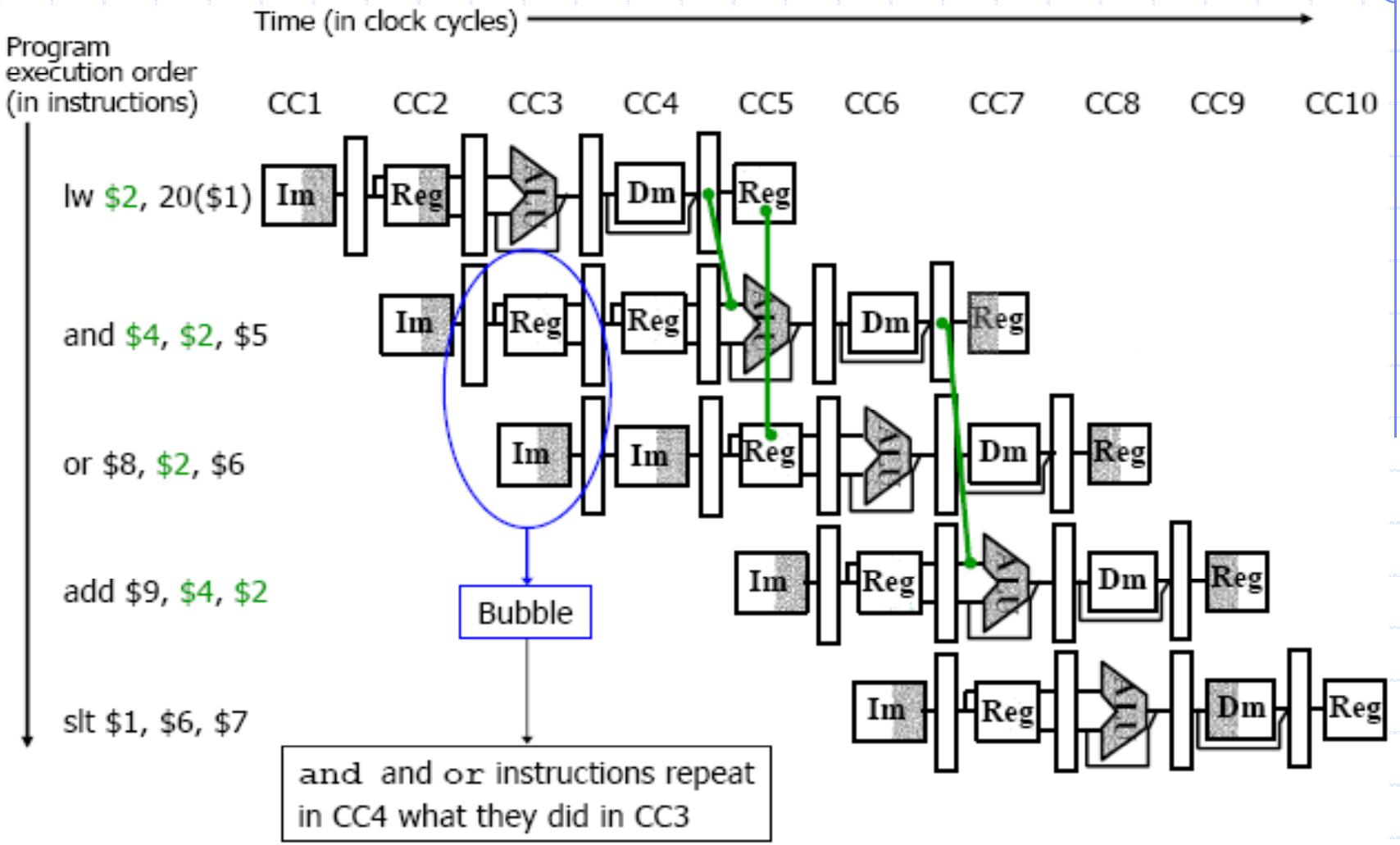
- کنترل برای کشف هزارد

چک کن اگر دستورالعمل بار کردن است

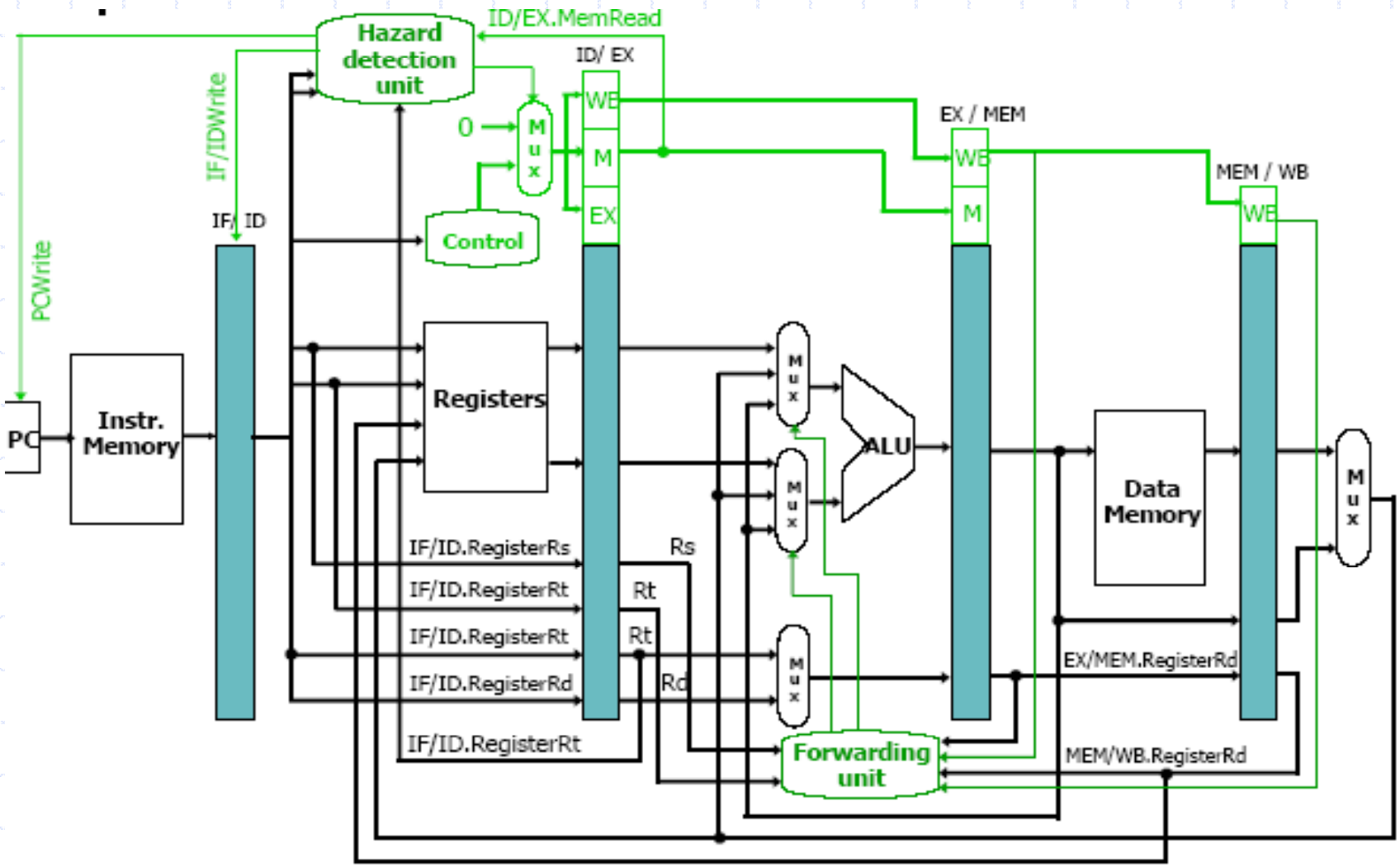
```
if (ID/EX.MemRd
and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)))
```

سپس:
توقف خط لوله

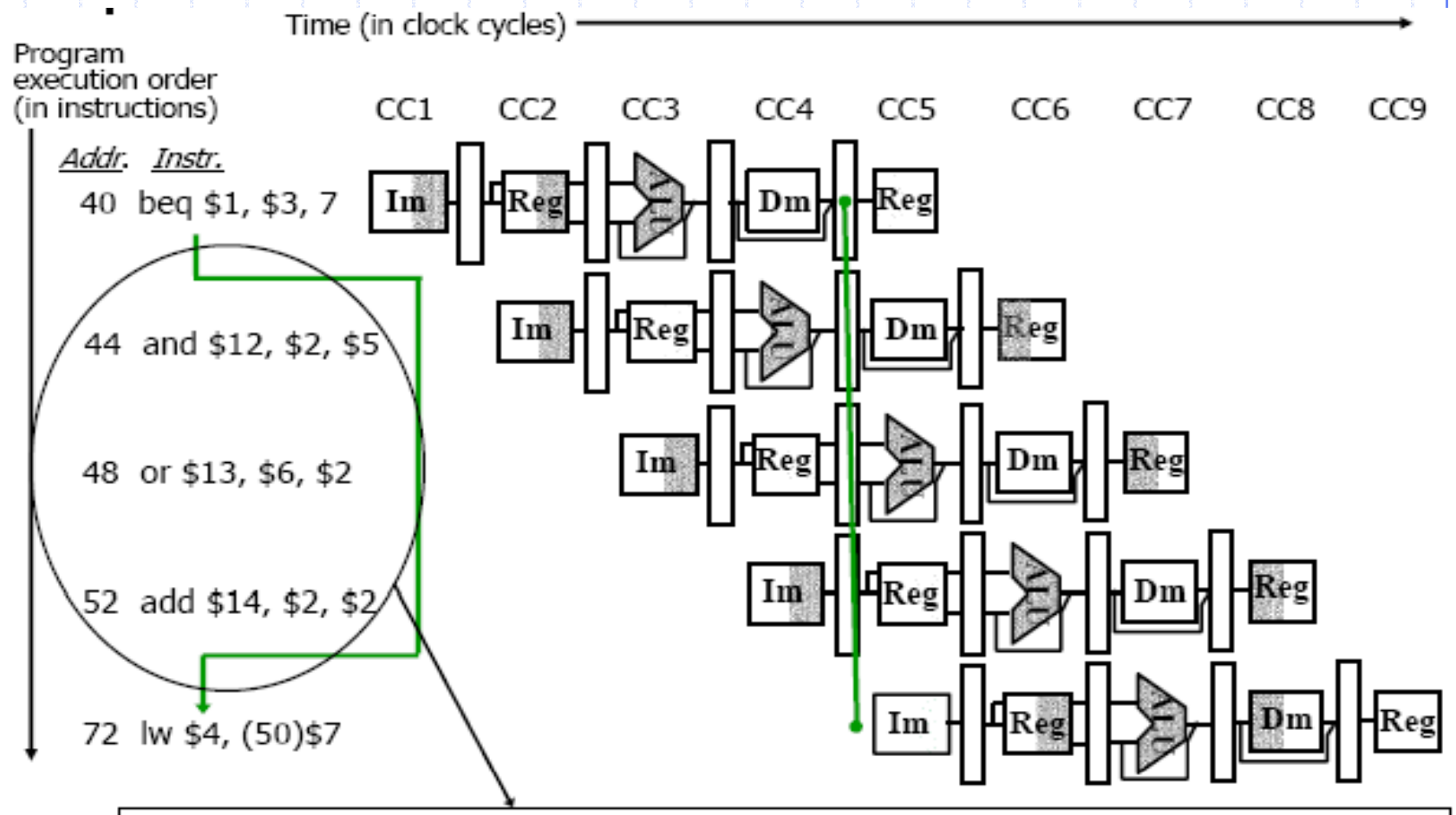
هزاردهای داده ای و توقف



مسیر داده خط لوله ای شده با(Forwarding)ارسال و واحد کشف هزارد

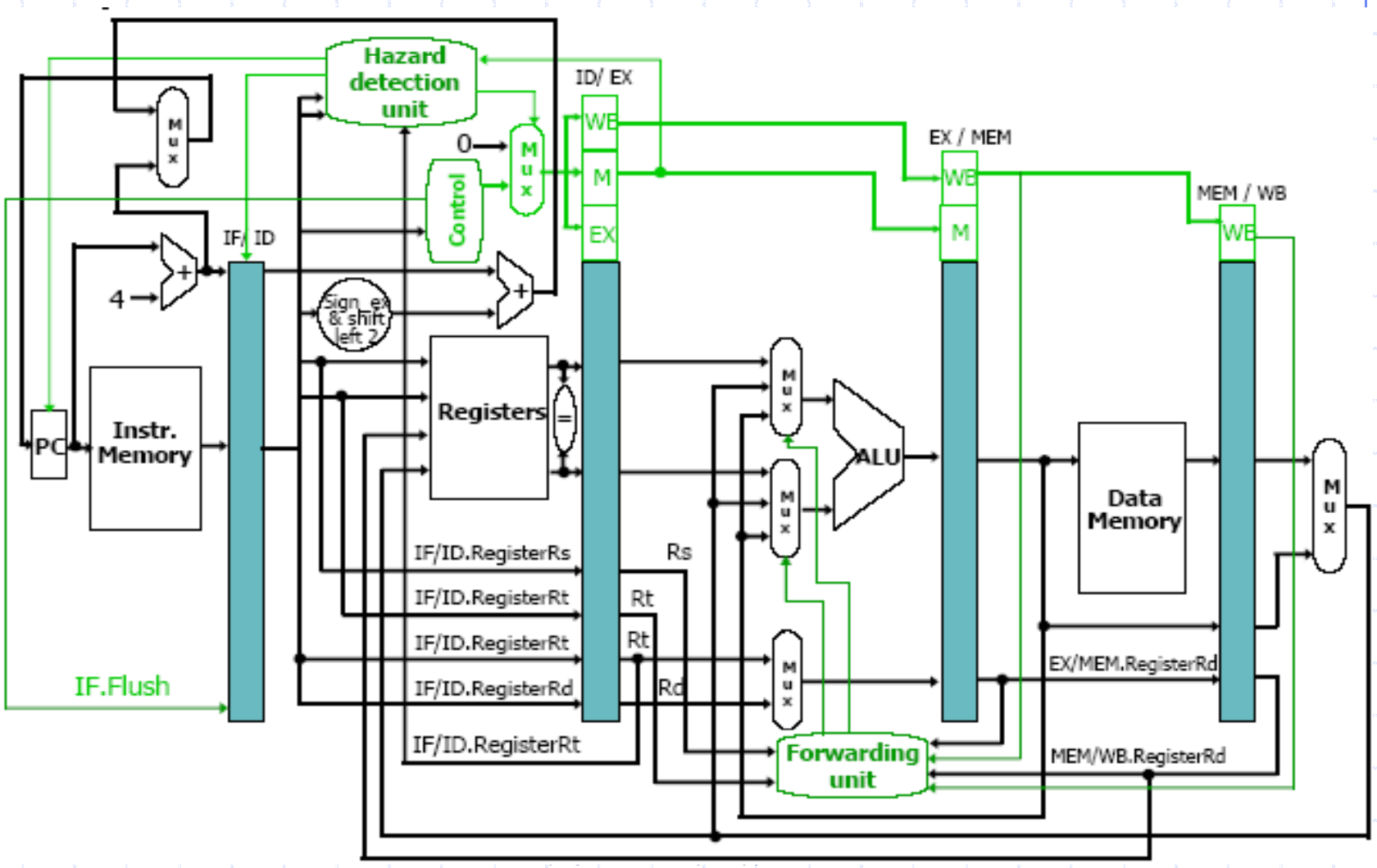


هزاردهای انشعاب



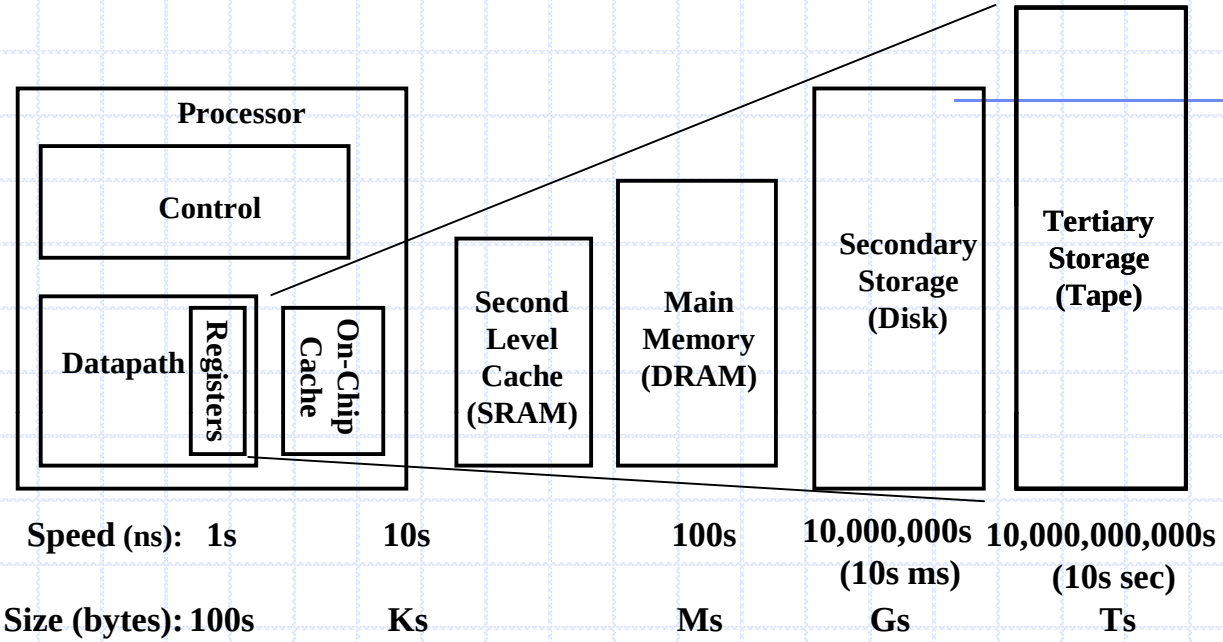
Actually, the number of instructions needs to be flushed can be reduced from 3 to 1 instruction (shown in the following slide) when the direction of branch is mispredicted.

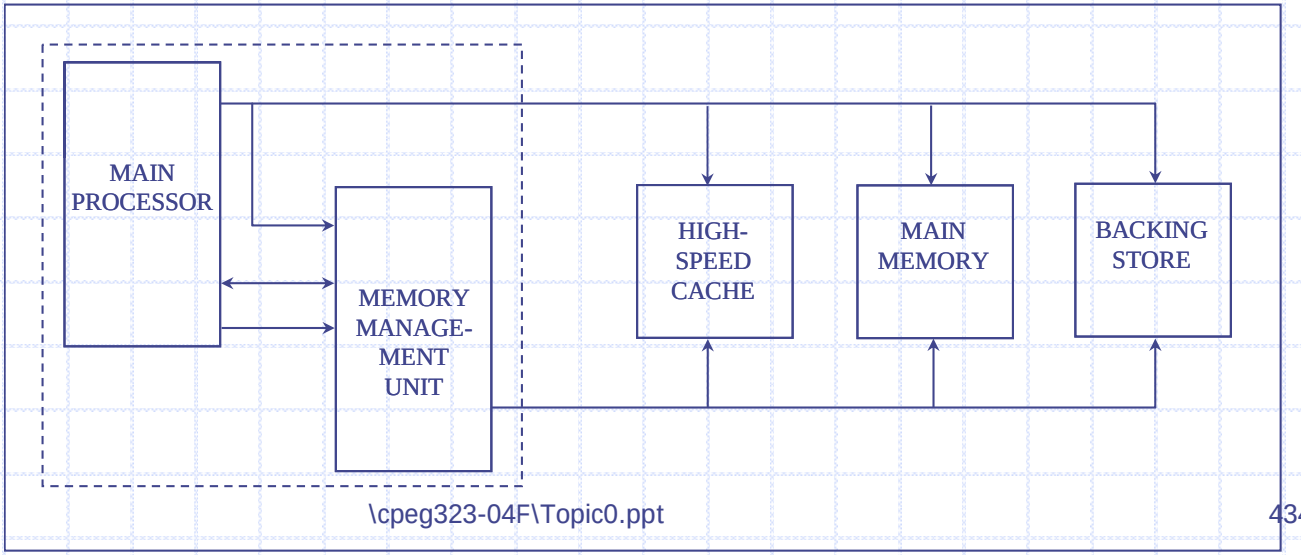
مسیر داده برای انشعاب(شامل HWخالی کردن قسمتی از حافظه خط لوله)



فصل هفتم
طراحی سیستم حافظه

سلسله مراتب حافظه

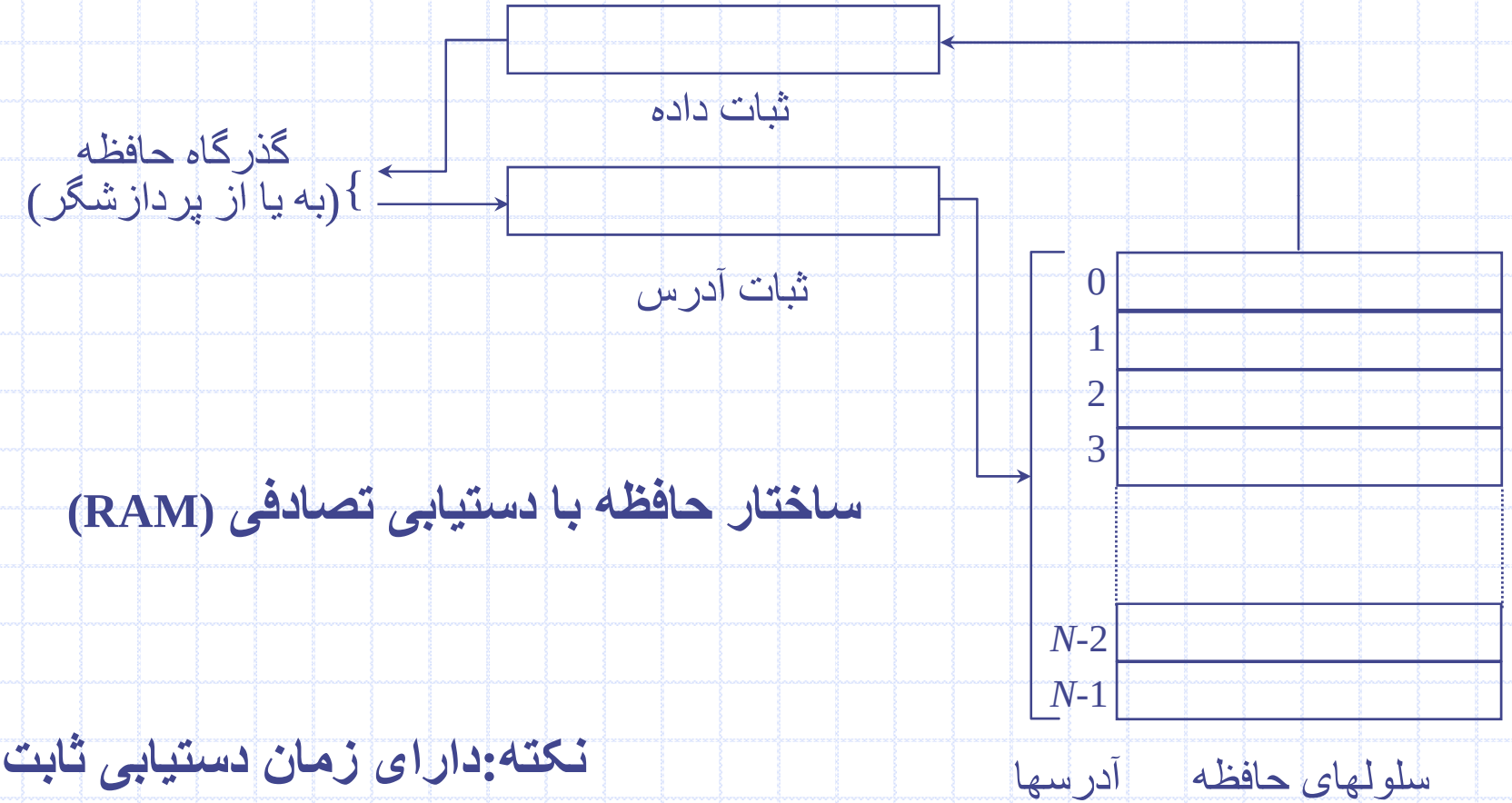




مشخصات برنامه ها و سازماندهی حافظه

- تجزیه و تحلیل Ram در برابر دستیابی ترتیبی:
  - در مقایسه **کارایی برهزینه** و **تکنولوژی**
- همجواری در الگوهای دسترسی به حافظه
- سلسله مراتب در طراحی حافظه
  - حافظه نهان
  - حافظه مجازی

حافظه با دستیابی تصادفی



ساختار حافظه با دستیابی تصادفی (RAM)

نکته:دارای زمان دستیابی ثابت

کارایی حافظه

پهنای باند = تعداد بیتها بر ثانیه "که می توانند در دسترس قرار گیرند"
که برابر است با:

(bit/word) x (word/cycle) x (cycle/sec)

بنابراین، بهبود پهنای باند؟

"گلوگاه ون نیومن"

چگونه می توان کارایی سیستم حافظه را بهبود بخشید

- کاهش مدت چرخه
- افزایش اندازه کلمه
- همزمانی
- طراحی کارآمد حافظه

حافظه نهان

- تقریباً تمام ریز پردازنده های با کارایی بالا در بازار از حافظه نهان استفاده می کنند.
- چرا معماری برداری (نوعی کامپیوتر بسیار بزرگ) Cray استفاده نشود؟

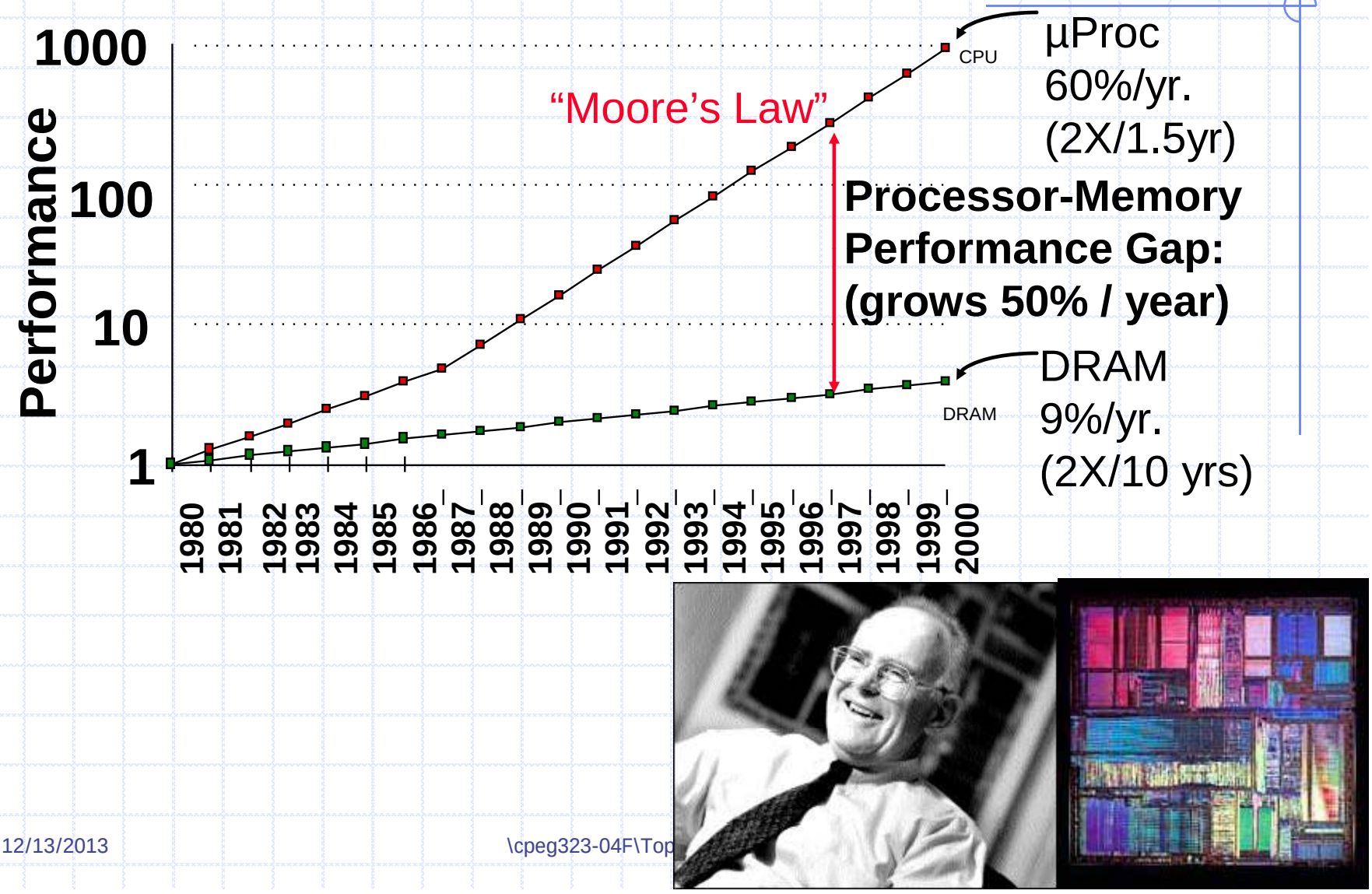
حافظه نهان

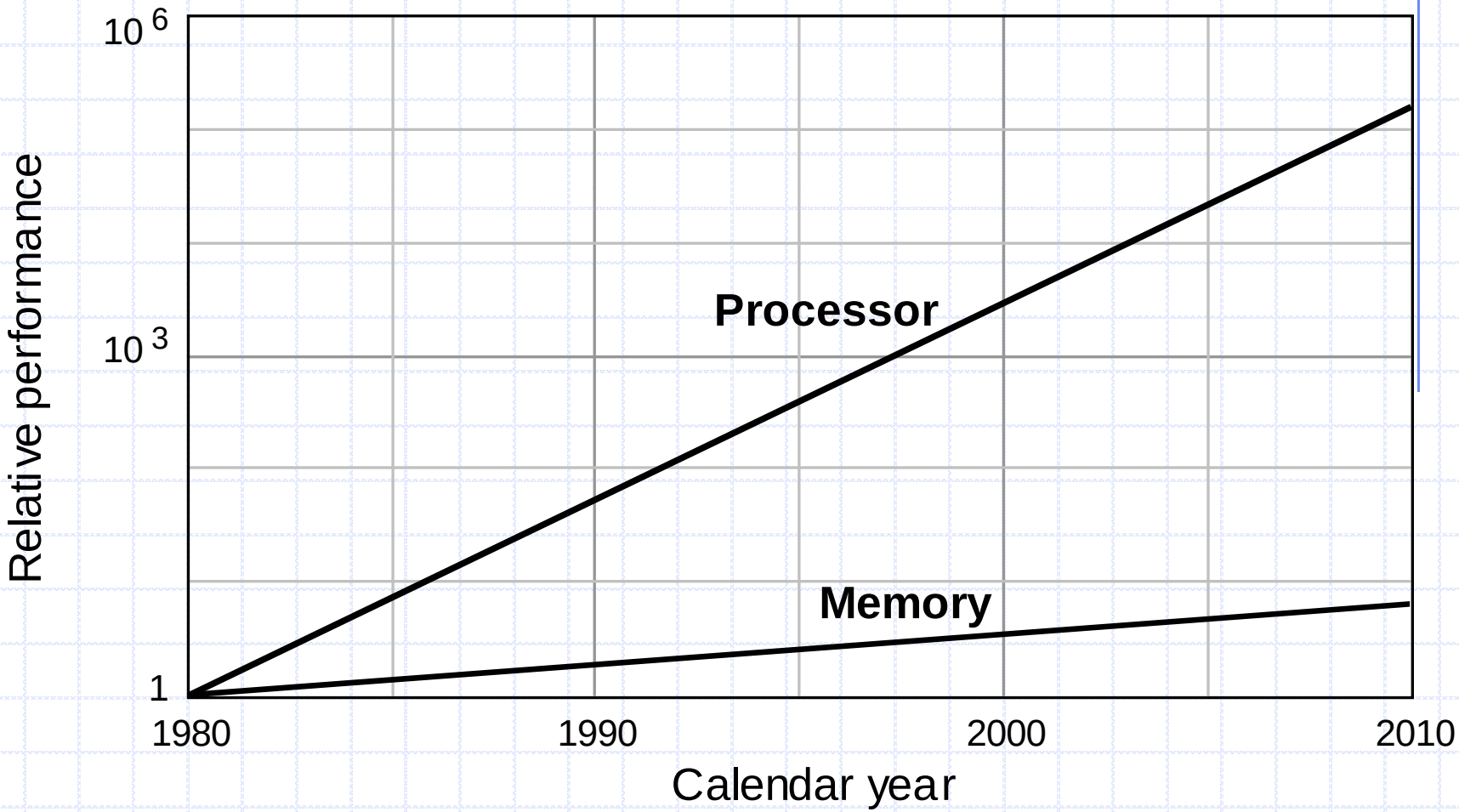
پیشرفتها در تکنولوژی مدارات مجتمع نه فقط متاثر از DRAM ها، بلکه SRAM ها، هرینه حافظه نهان را بسیار کاهش داده اند. حافظه های نهان یکی از ایده های بسیار مهم در معماری کامپیوتر هستند زیرا آنها می توانند کارایی استفاده از حافظه را به طور اساسی بهبود بخشند.

افزایش شکاف بین زمان چرخه DRAM و زمان چرخه پردازنده، که در شکل بعد نمایش داده شده است، یک انگیزه برای ایجاد حافظه نهان است. اگر ما بر آنیم که پردازنده ها با سرعتی که توانایی آن را دارند اجرا شوند، ما باید حافظه هایی با سرعت بیشتر برای فراهم نمودن داده داشته باشیم.

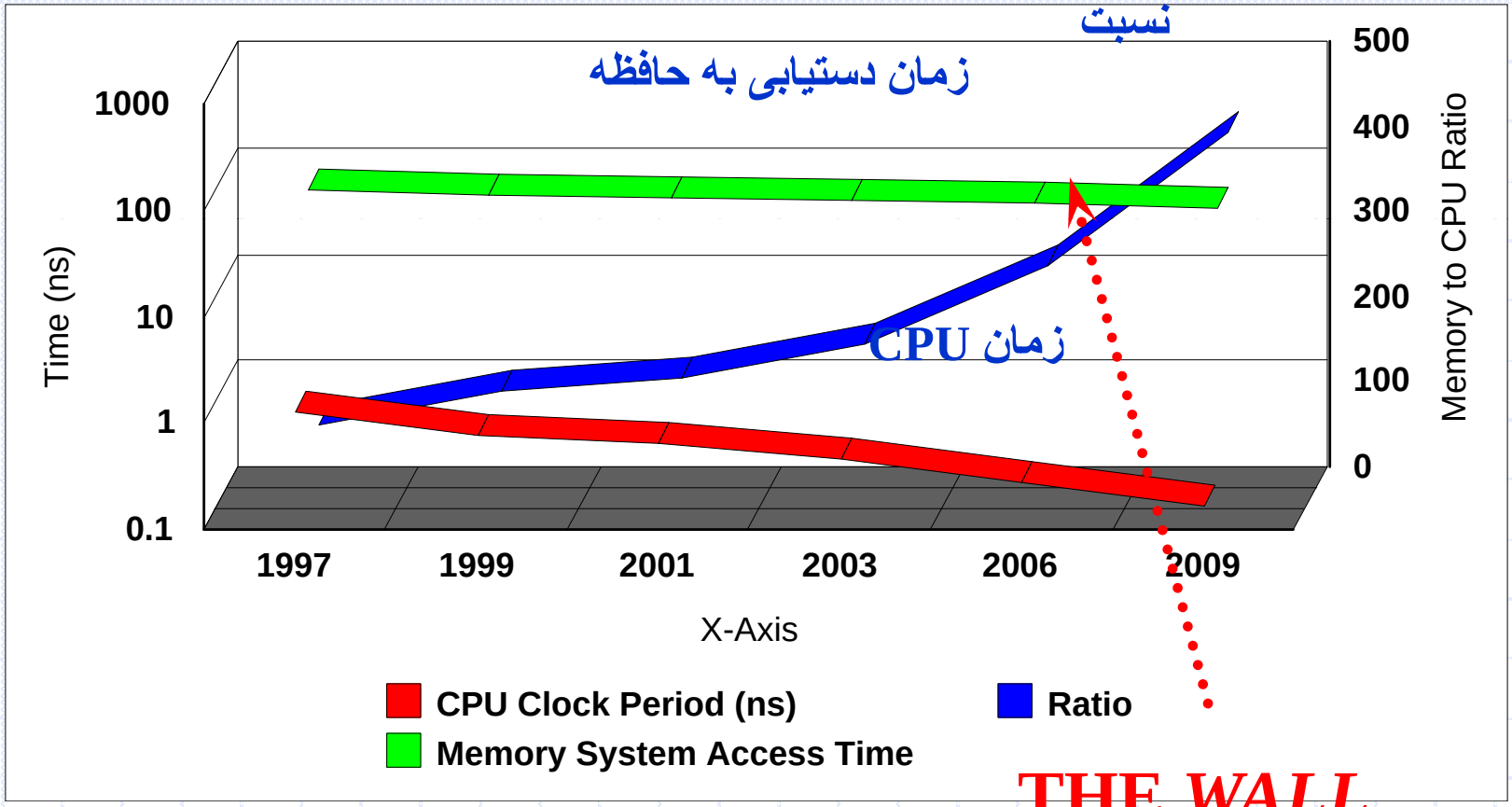
[Joupi & Hennessy 91]

قانون مور – یک فرصت از دست رفته




Relative performance
10 6
10 3
1
Processor
Memory
1980
1990
2000
2010
Calendar year

نهفتگی در یک سیستم واحد

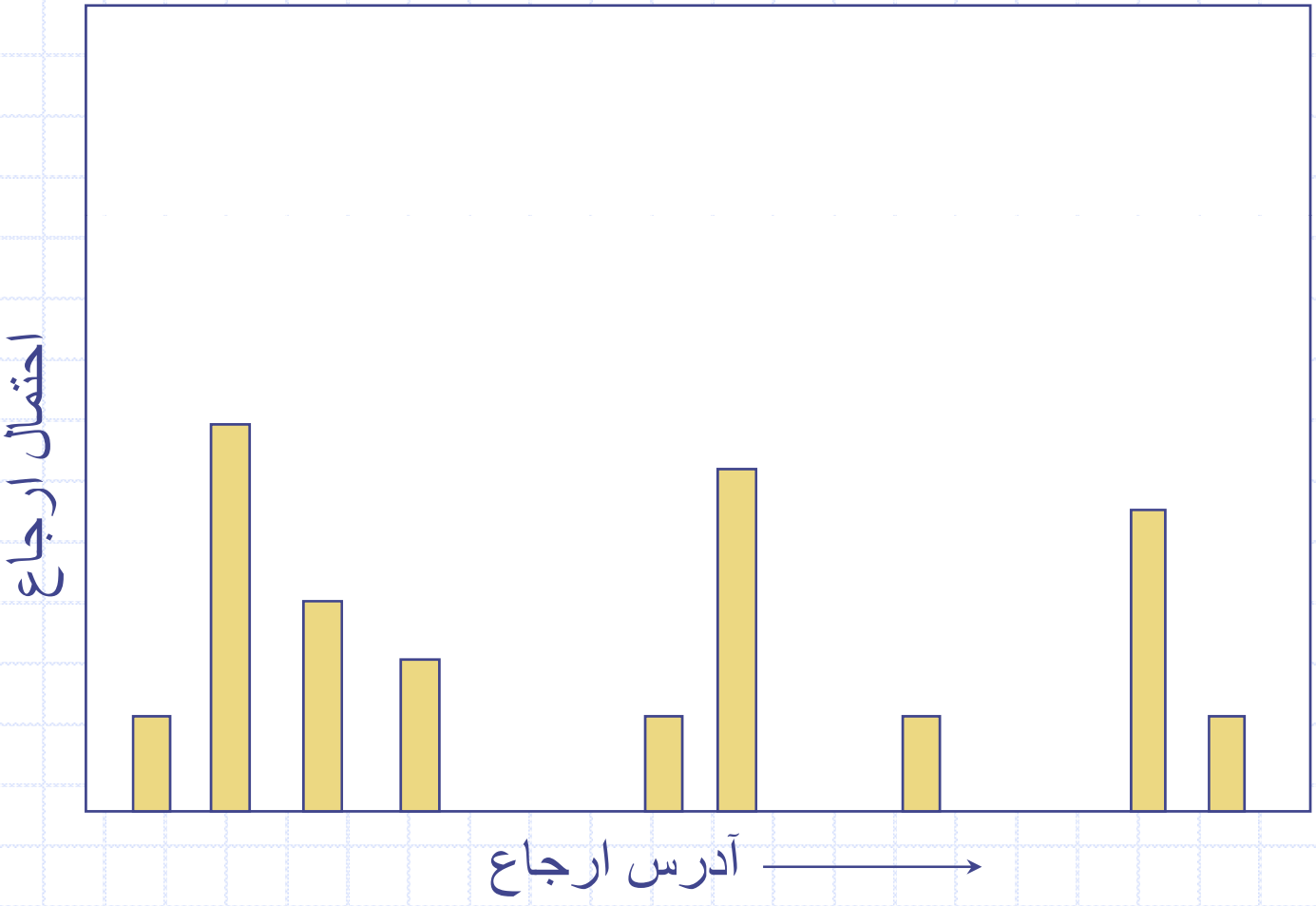


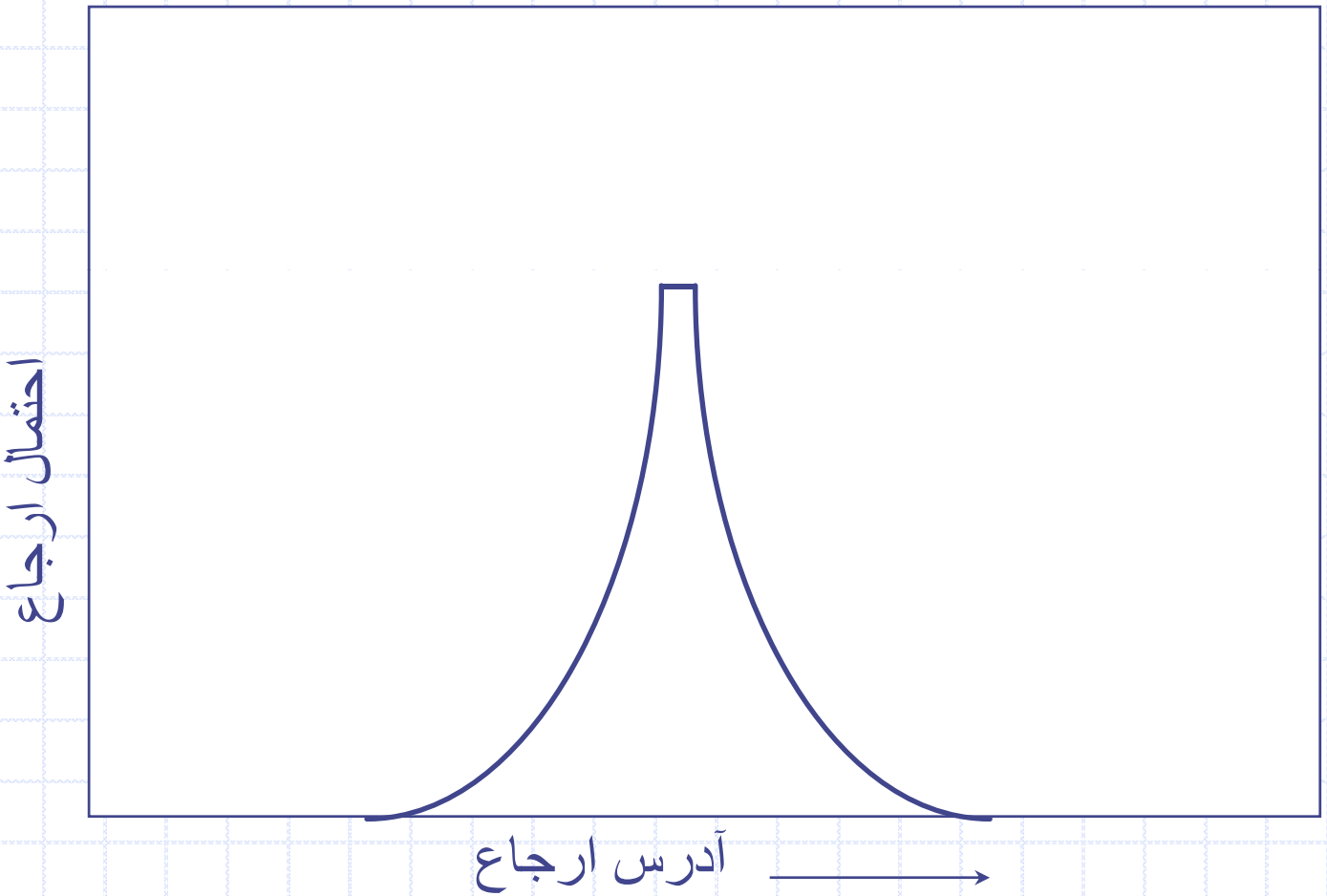
همجواری مراجع

“ مراجع برنامه تمایل دارند که به موقع در کنار هم قرار بگیرند”.

مکانهایی با احتما ل دستیابی بالا (زمان کوتاه)

- همجواری PC
- قاب پشته (محلی)
- نزدیک زیر روال ها
- داده های فعال


احتمال ارجاع
آدرس ارجاع



مشکل: اغلب پیش بینی یک page sizeمی شود که بزرگتر ازاندازه مورد نیاز باشد.

همجواری

موفقیت حافظه های نهان که قبلا تشریح شد برگرفته از "خاصیت همجواری" است.

خاصیت هجواری دارای دو جنبه است، *temporal* و *spatial*.

در دوره های کوتاه زمانی، یک برنامه ارجاعها به حافظه خود را، به صورت غیر یکنواخت در فضای آدرس خود گسترش می دهد.

که این بخشهای فضای آدرس در دوره های زمانی طولانی به صورت گسترده یکنواخت باقی می مانند.

همجواری موقتی

اولین خاصیت، معروف به همجواری موقتی، یا همجواری زمانی، به این معنی است که: اطلاعاتی که در آینده نزدیک مورد نیازند احتمالاً هم اکنون در حال استفاده اند.

این نوع رفتار می تواند از حلقه های برنامه که هر دوی داده و دستورالعمل مجدداً استفاده می شوند، انتظار رود.

همجواری مکانی

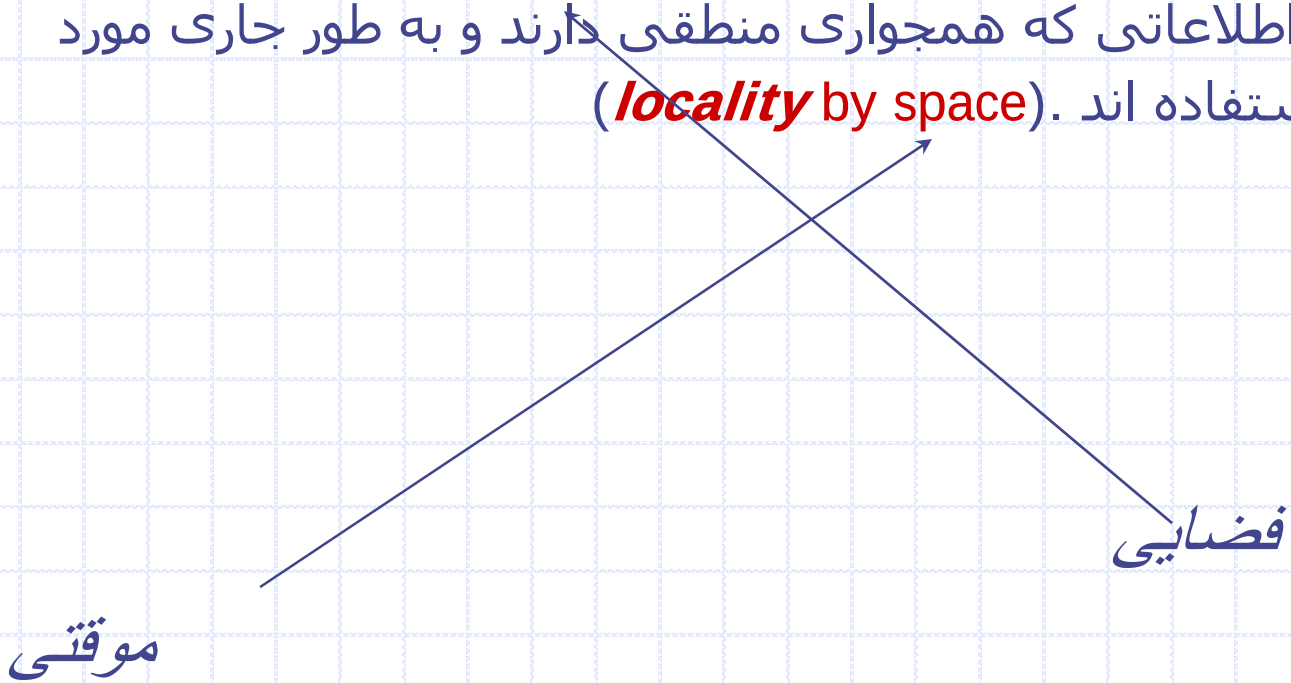
دومین خاصیت، همجواری مکانی، به این معنی است که بخشهایی از فضای آدرس که در حال استفاده اند معمولاً شامل یک تعدادی از بخشهای همجوار یکتا از فضای آدرس می باشد. همجواری مکانی، سپس،به این معناست که محل ارجاع برنامه در آینده نزدیک، احتمالاً نزدیک محل ارجاع فعلی است

[Smith82, p475]

این نوع رفتار با آگاهی معمولی از برنامه ها مورد انتظار است؛ بخشهای وابسته داده (متغیرها، آرایه ها) معمولاً با یکدیگر ذخیره می شوند، و دستورات بیشتر به صورت ترتیبی اجرا می شوند. تا زمانیکه حافظه نهان سگمنتهای اطلاعاتی را که اخیراً استفاده شده اند،میانگیر (در خود نگهداری)می کند،خاصیت همجواری اشاره دارد به اینکه اطلاعات مورد نیاز احتمالاً در حافظه نهان یافت می شود.

اطلاعاتی که در آینده نزدیک مورد نیازند احتمالاً شامل اطلاعات رایجی هستند که هم اکنون در حال استفاده اند. (***locality*** by time)

و آن اطلاعاتی که همجواری منطقی دارند و به طور جاری مورد استفاده اند .(***locality*** by space)



مقدمه ای بر طراحی cache

(نوعي حافظه ميانجي كه سرعت دستيابي به اطلاعات آن بالا است)

Cache(حافظه نهان)

مکانی امن برای پنهان سازی و ذخیره سازی.

Webster Dictionary

Cache(حافظه نهان)

- حتی با وجود ظرفیت cache هنوز تقریباً همه cpu ها شدیدا به وسیله زمان دستیابی cache محدود شده اند.
در بسیاری از گونه ها، هرچند زمان دستیابی cache کاهش داده شده باشد ماشین نیز باید تسریع شود.

-Alan Smith-

Even more so for MPs!

- الگوهایی که می توانند طراحی حافظه cache حاضرراشکست دهند، باعث افزایش کارایی حافظه cache شده،وبرای هر برنامه یا بارکاری واقعاً محاسبات مفیدی انجام دهد.

- بهینه کردن طراحی حافظه cacheبه طور گسترده که دارای 4 جنبه است:

1. به حد اعلی رساندن احتمال یافتن یک مرجع حافظه در cache(نسبت برخورد)
2. به حداقل رساندن زمان دستیابی به اطلاعاتی که در cache وجود دارند(زمان دستیابی)
3. به حداقل رساندن تأخیر به علت یک خطا و
4. به حداقل رساندن هزینه های ناشی از به روز کردن حافظه اصلی،نگهداری ***cache coherence*** و غیره.

فاکتورهای کلیدی در تصمیم طراحی برای CacheوVM (حافظه مجازی)

$$\frac{\text{زمان دستیابی حافظه اصلی}}{\text{زمان دستیابی cache}} \doteq 4 \sim 20$$

$$\frac{\text{زمان دستیابی حافظه ثانویه}}{\text{زمان دستیابی حافظه اصلی}} \doteq 10^4 \sim 10^6$$

کنترلcache معمولاً در سخت افزار اجرا می شوند

فن آوری در1990s:

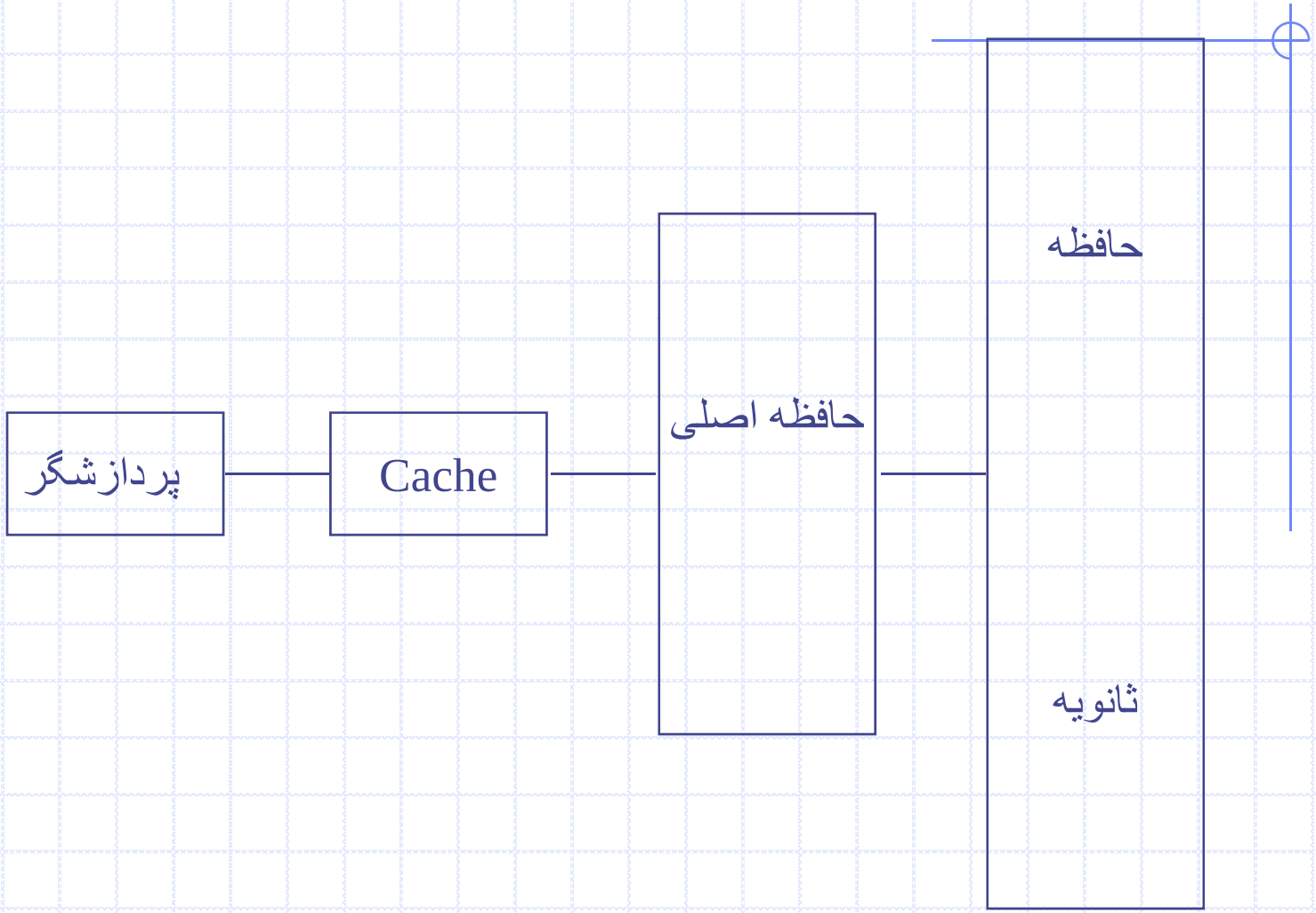
Memory Technology	Typical Access Time	$ per Mbyte
SRAM	10-20 ns	200-400
DRAM	90-120 ns	50-100
Magnetic disk	10,000,000 - 20,000,000 ns	2-5

فن آوری در 2000s:؟

فنآوری در2000s :

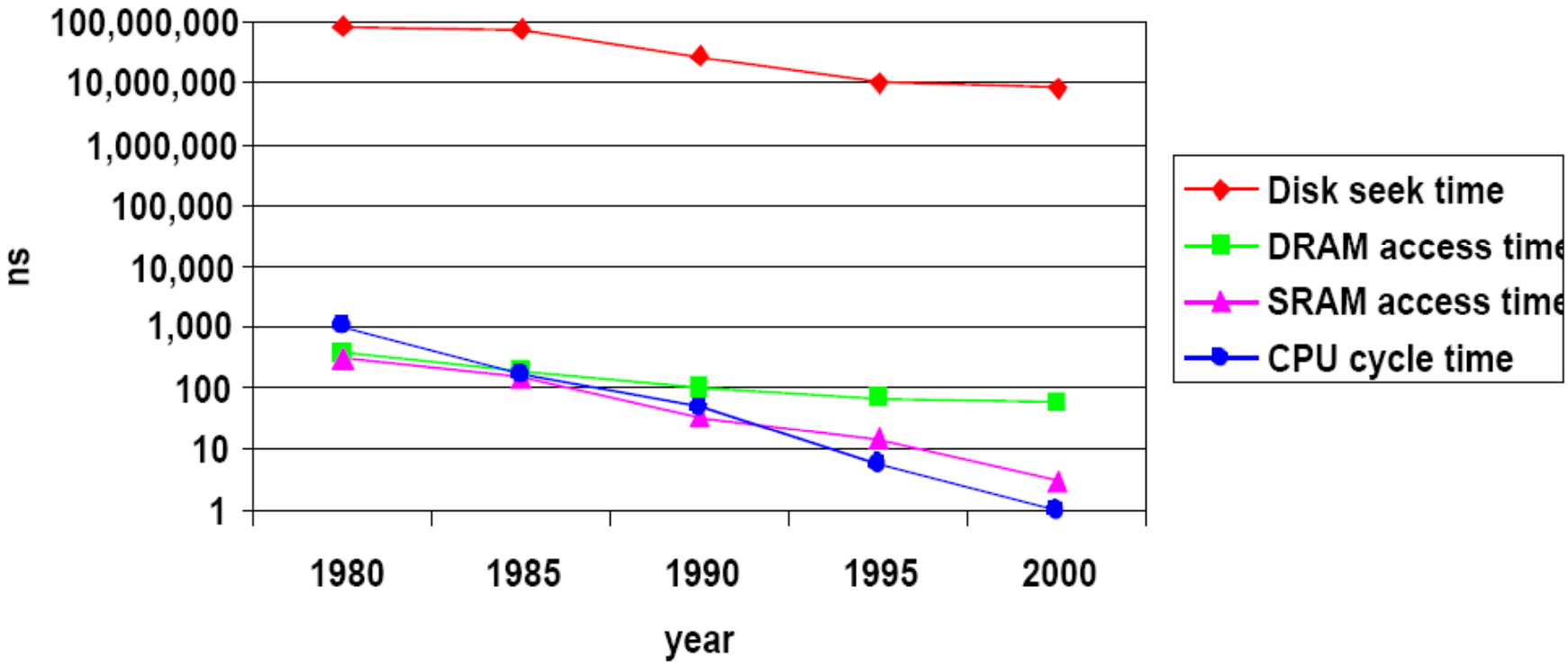
Memory Technology	Typical Access Time	$ per Mbyte
SRAM	5-10 ns	1-20
DRAM	10-60 ns	0.1-1
Magnetic disk	2,000,000 - 5,000,000 ns	0.001-0.006

Cacheدرسلسله مراتب حافظه



The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.



چهار سوأل برای دسته بندی سلسله مراتب حافظه

قاعده کلی و اساسی که گرداننده همه سلسله مراتب های حافظه است به ما اجازه می دهد از شرایطی که باعث برتری یافتن نسبت به سطوحی که در موردشان سخن گفتیم استفاده کنیم.

این قواعد یکسان به ما اجازه طرح چهار سوأل درباره هر سطح از سلسله مراتب را می دهد:

چهار سوأل برای دسته بندی سلسله مراتب حافظه

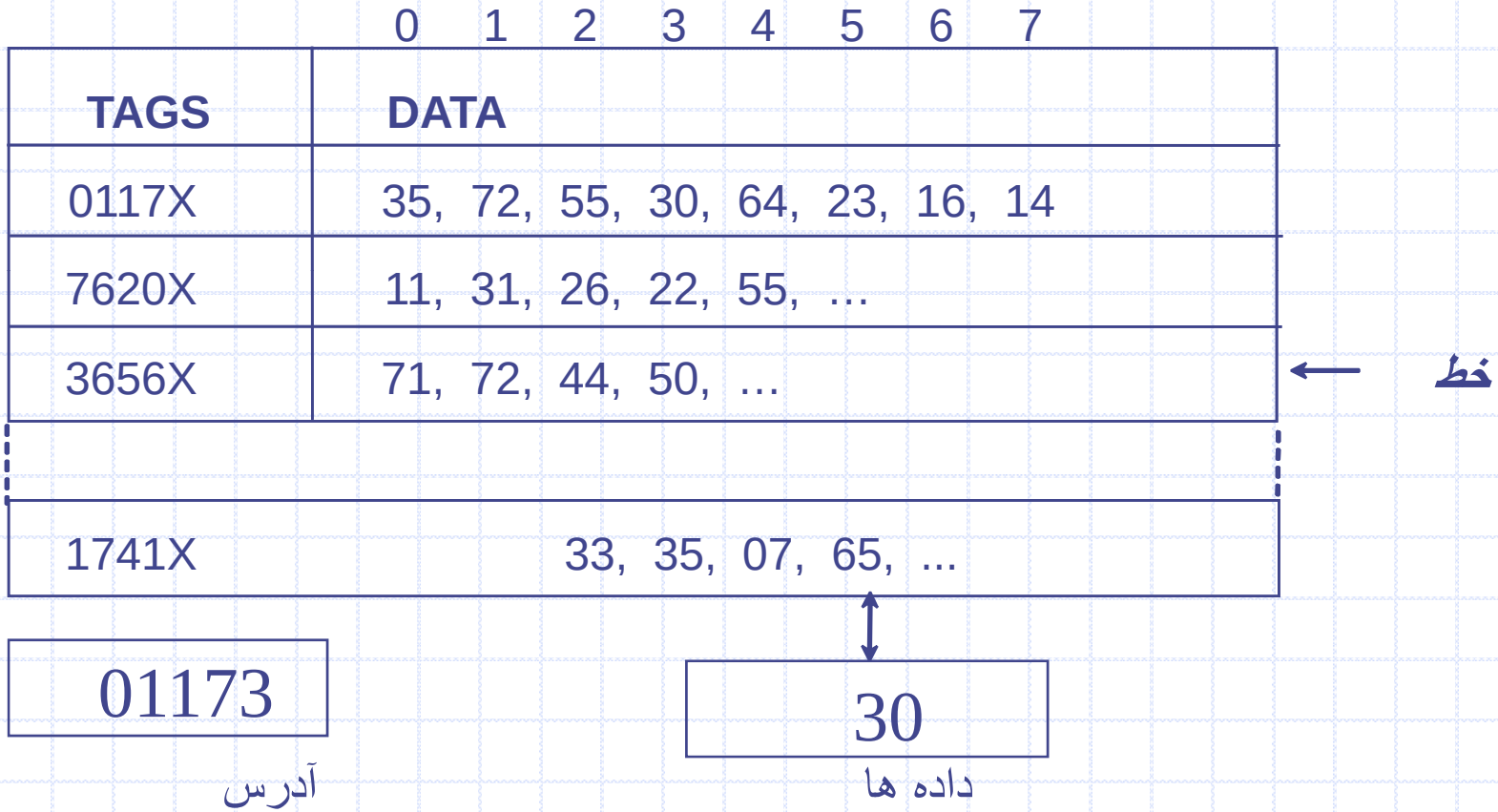
س1: *در کجا یک بلاک می تواند در سطح بالاتری قرار بگیرد؟* (مکان بلاک)

س2: چگونه وقتی یک بلاک در سطح بالاتری قرار دارد شناخته می شود؟(*هویت بلاک*)

س3: کدام بلاک می تواند در یک فقدان جایگزین شود؟ (جایگذاری بلاک)

س4: در نوشتن چه روی می دهد؟ (استراتژی نوشتن)

این سوألات ما را در جهت رسیدن به یک درک کامل از مبادلات
متفاوت در خواست شده به وسیله ارتباطات حافظه ها در سطوح
متفاوت یک سلسله مراتب کمک می کند.



مفهوم cache miss وcache hit

t_{eff} : زمان دستیابی موءثر cache

t_{cache} : زمان دستیابی cache

t_{main} : زمان دستیابی حافظه اصلی

h : نسبت برخورد(موءفقیت)

$$t_{eff} = ht_{cache} + (1-h)t_{main}$$

مثال

Let

t_{cache} = 10 ns - 1 - 4 clock cycles

t_{main} = 50 ns – 8 - 32 clock cycles

h = 0.95

t_{effect} = ?

10 x 0.95 + 50 x 0.05

9.5 + 2.5 = 12

نسبت برخورد(یا موءفقیت)

- نیاز به نیروی کافی برای رسیدن به سطح کارایی دلخواه دارد(say>90%)
- وسعت بخشی به اثرات و نتایج تغییرات
- هرگز ثبات دائمی حتی برای ماشین های مشابه ندارد

حساسیت کارایی *w.r.th* (hit ratio)

$t_{eff} = h\, t_{cache} + (1-h)\, t_{main}$

$t_{eff} = t_{cache} [\, h + (1-h) \frac{t_{main}}{t_{cache}} \,]$

$t_{cache} [\, 1 + (1-h) \frac{t_{main}}{t_{cache}} \,]$

$\frac{t_{main}}{t_{cache}} \approx$ 10, since the magnifactor of h changes is 10 times.

نتیجه:خیلی حساس

- یادآوری

“h 1” $\approx$

- مثال:

Let $h = 0.90$

if $h \triangleq 0.05$ $(0.90 \longrightarrow 0.95)$

then $(1 \triangle h) = 0.05$

then $t_{eff} = t_{cache} (1 + 0.5)$

اصطلاحات بنیادی

- Cache line (block) - size of a room

  1 ~ 16 words

  “ یک مجموعه از داده های همجوار که به عنوان یک نهد ذخیره سازی cache مورد بحث واقع شده اند”.

- Cache directory - key of rooms

  بخشی از cache که کلیدهای دستیابی را که دستیابی اشتراکی را پشتیبانی می کنند نگهداری می کنند

cache باید مشارکت دز پیدا کردن “راهنما ی درست”به وسیله تطبیق داشته باشد

سازمان دهی cache

- کاملاً متحد: می تواند یک عنصر درهر بلاک باشد
- مسیر دهی بی واسطه و مستقیم: می تواند یک عنصر فقط دریک بلاک باشد
- چیدمان و قرارگیری پیوسته: می تواند یک عنصر درگروهی از بلاک ها باشد

پیوستگی کامل

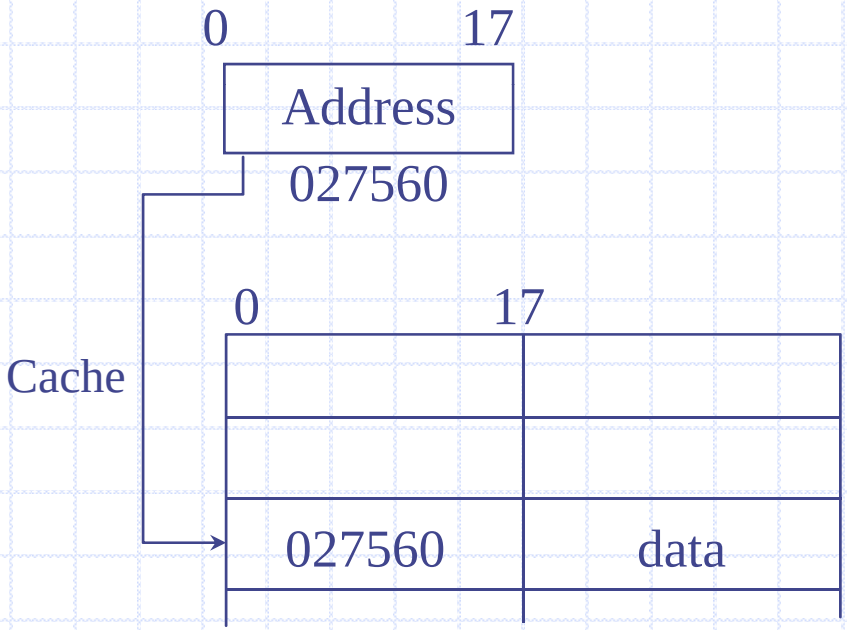
ساختار

- هربلاک در حافظه می تواند در هر block-frameدر cacheباشد
- تمام ورودی های الگوی بلاک (block frame)باهم مقایسه می شوند(به وسیله جستجوی پیوسته)

ساده ترین مثال: یک بلاک = a word

تمام حافظه آدرس دهی کلمه ای شدن

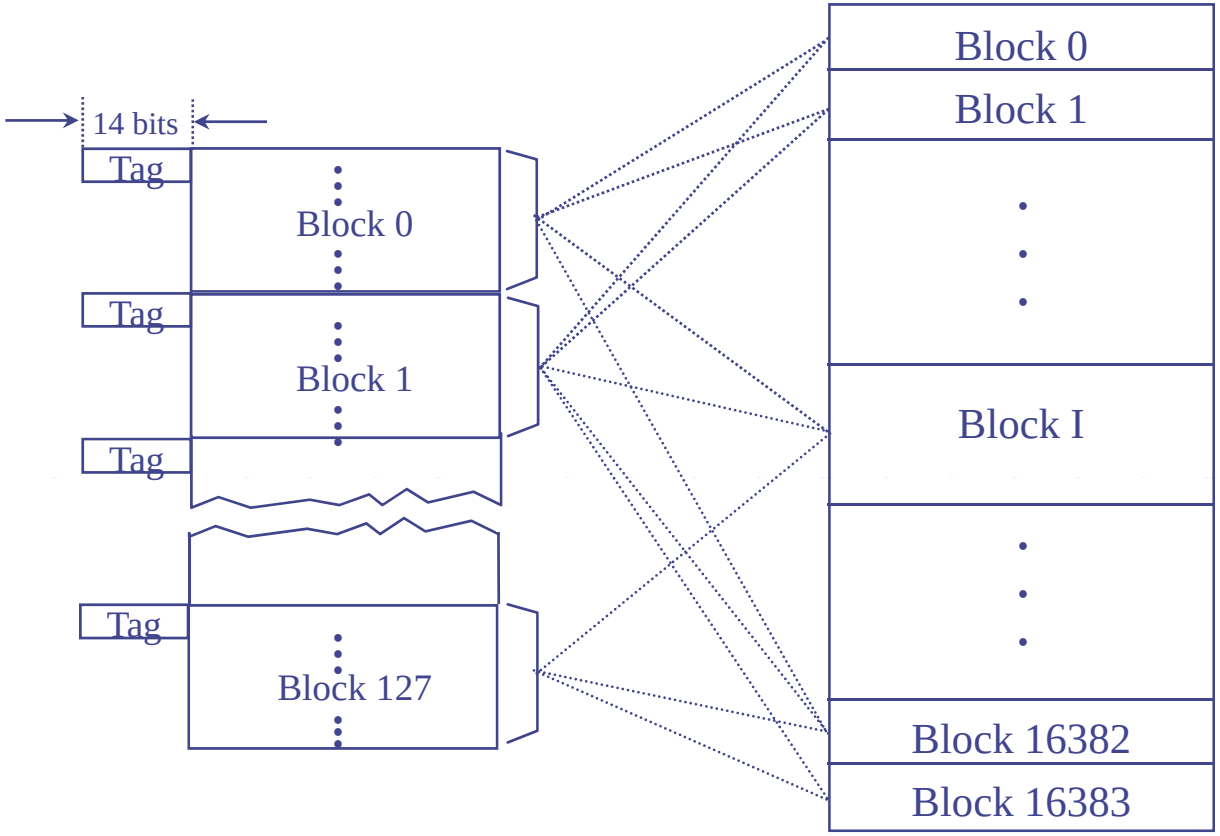
"tag"



بسیار انعطاف "وبا احتمال بالا به منظور ساکن شدن در cache

مزایا: بدون اتلاف (با قابلیت سازمان دهی سریع)

معایب: overhead of associative search:
هزینه+ زمان



یادآوری: هر بلاک 16 کلمه دارد

سازمان کاملاً پیوسته cache

نگاشت مستقیم

- بدون پیوند پیوسته
- از آدرسM حافظه"به طور مستقیم"برای قاب بلاک در cacheجایی که بلاک باید قرار بگیرد شاخص دار شده. A comparison then is to used to determine if it is a miss or hit.

نگاشت مستقیم

ساده ترین امتیاز:

سریع(با منطق کمتر)
هزینه کم:از این رو فقط یک دستگاه مقایسه کننده
لازم است که می تواند در فرم استاندارد حافظه قرار گیرد

معایب:اتلاف و بیهودگی

نگاشت مستقیم

- مثال:

زمانی که cache فقط 128 الگوی قاب دارد در نتیجه درجه تسهیم چنین می باشد:

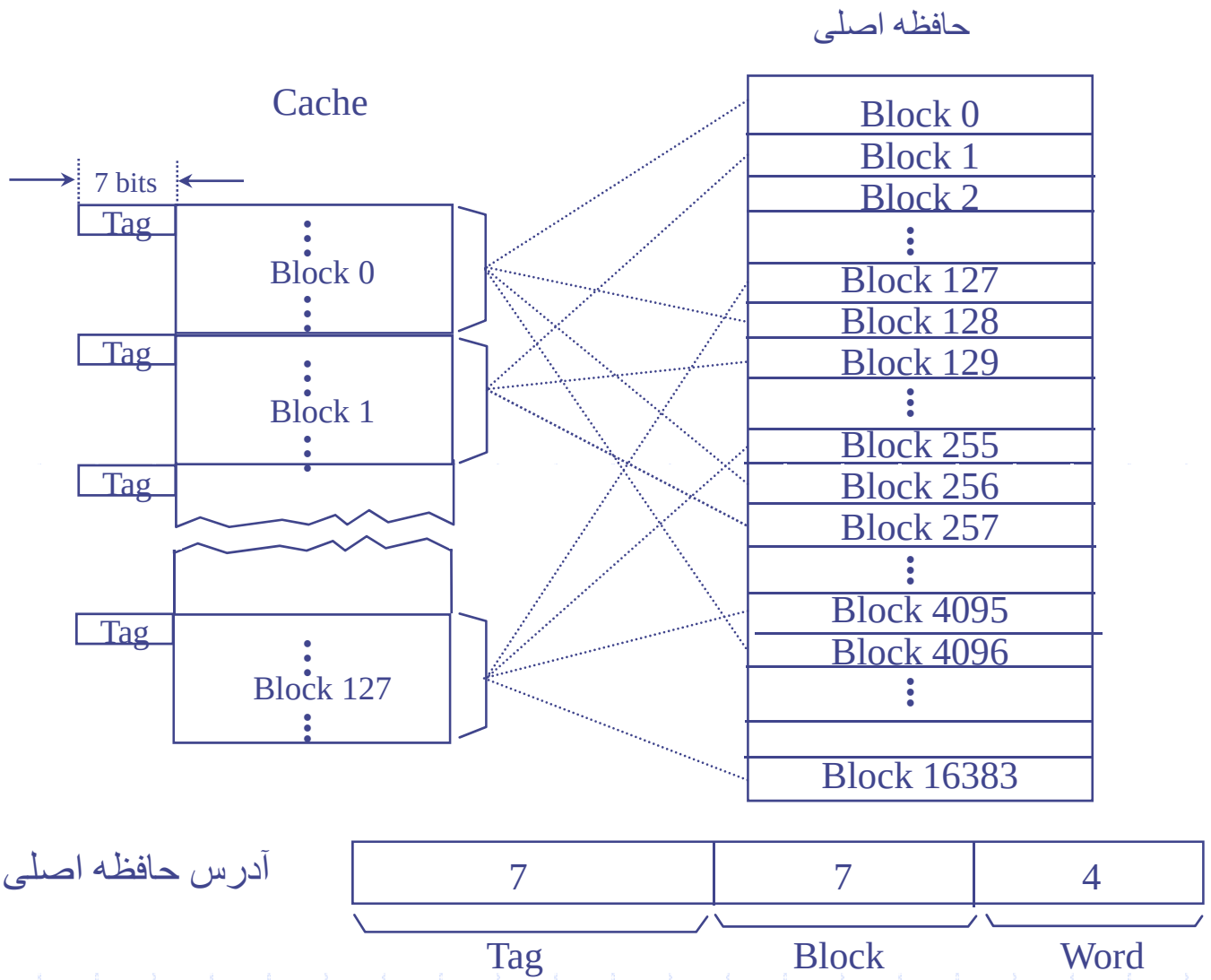
for addressing the corresponding frame or set of size 1.

$$\frac{\text{اندازه حافظه اصلی}}{128\ (2^7)} = \frac{16384\ (\text{block})}{128} = 2^7\ \text{block/frame}$$

the high-order 7 bit is used as tag.

i.e. 2^7 blocks “fall” in one block frame.

Disadr: “frashing”



نگاشت مستقیم

نگاشت مستقیم

Cont'd

Mapping block addr mod (# of blocks in cache – in this case: mod (2^7))

مزیت :مرتبه پایین $\log_2$ (اندازه cache)بیت می تواند برای شاخص دار شدن استفاده شود

قرار گیری پیوسته

- توافق بین direct/full-associative
- Cacheبه چیدمانهای s تقسیم می شود

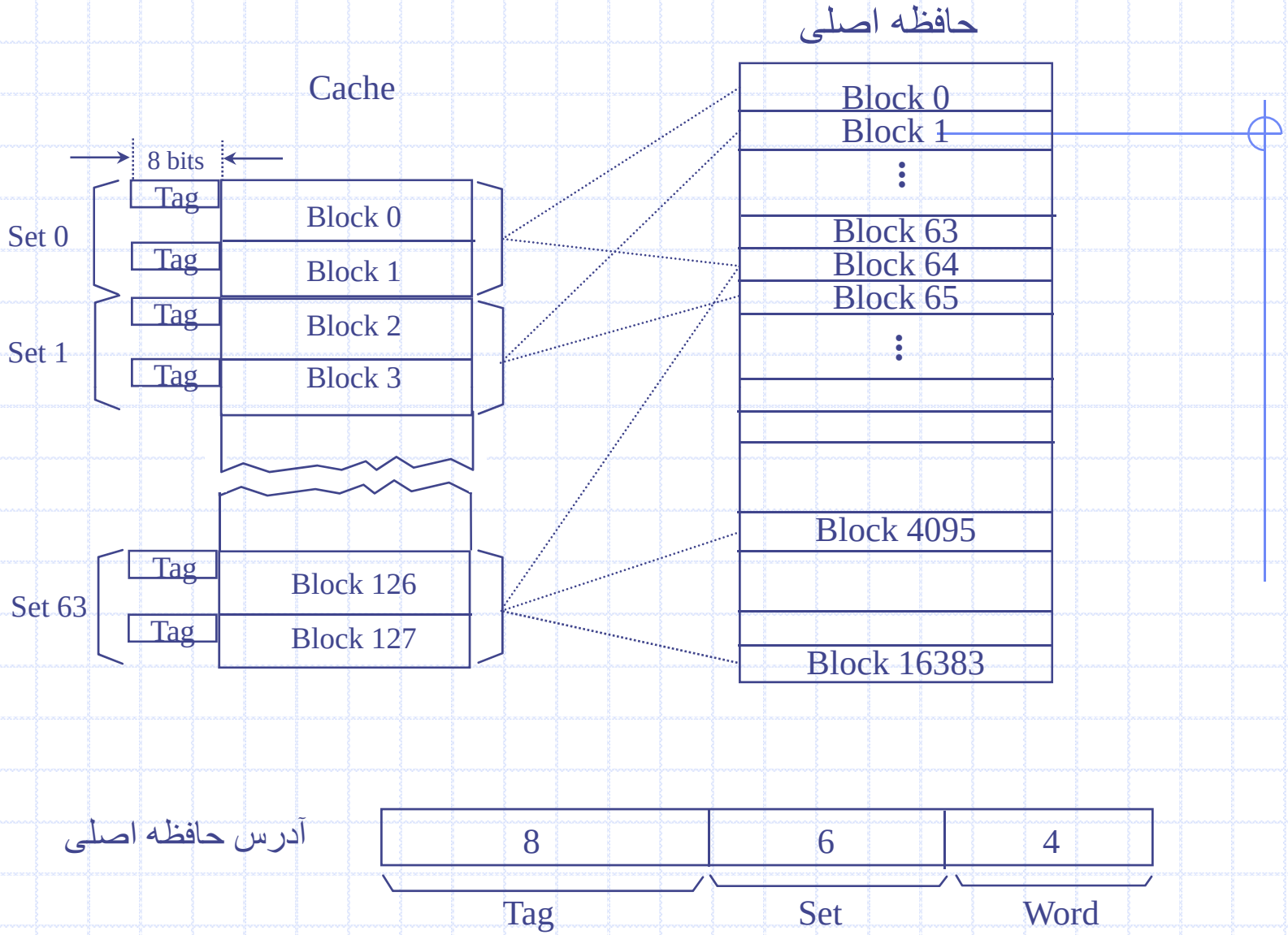
S = 2, 4, 8, …

- اگر M cache قاب داشته باشد که همه باهم باشند آنگاه داریم:

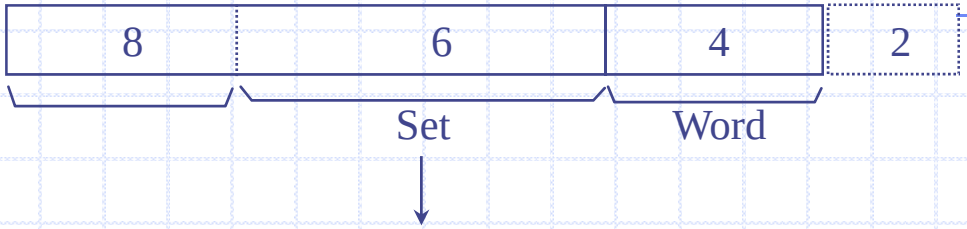
$$E = \frac{M}{S} \quad \text{blocks/set}$$

تعداد ساختارهای قابل دسترسی برای شاخص دار کردن

6 بیت می توانند در چیدمان سمت راست فهرست شوند و سپس 8 بیت برای پیوند پیوسته به کار خواهد رفت



2 راه برای قرار گرفتن سازمان پیوسته



آماده دسترسی برای
شاخص دار کردن

6 bit used to index into the right "set" higher order

$$\frac{2^{14}\,(16k)}{2^{6}} = 2^{8} \text{ block/set}$$ thus

2 way

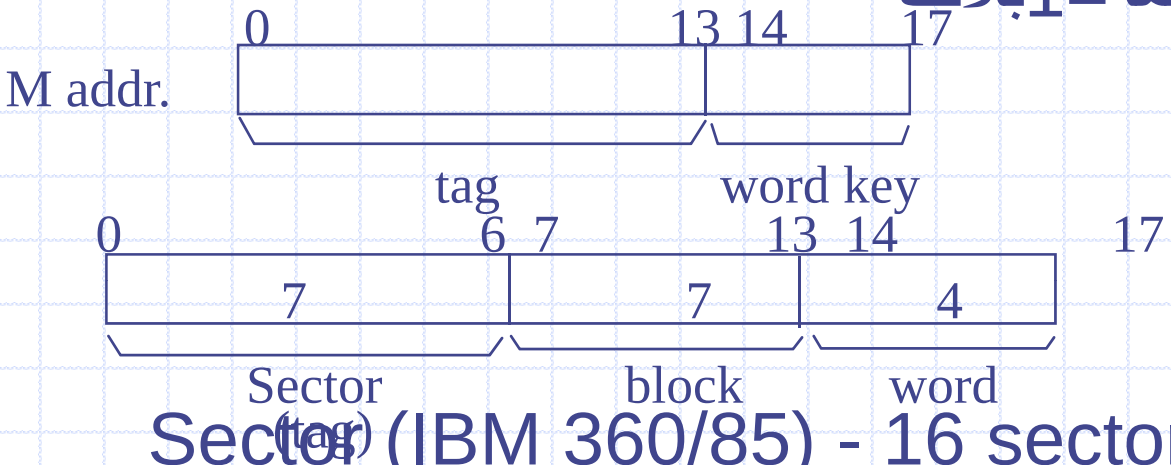
2^{8} block/per set of 2 or blocks

8 bit used as tag hence an associative match of 8 bit with the tags of the 2 blocks is required

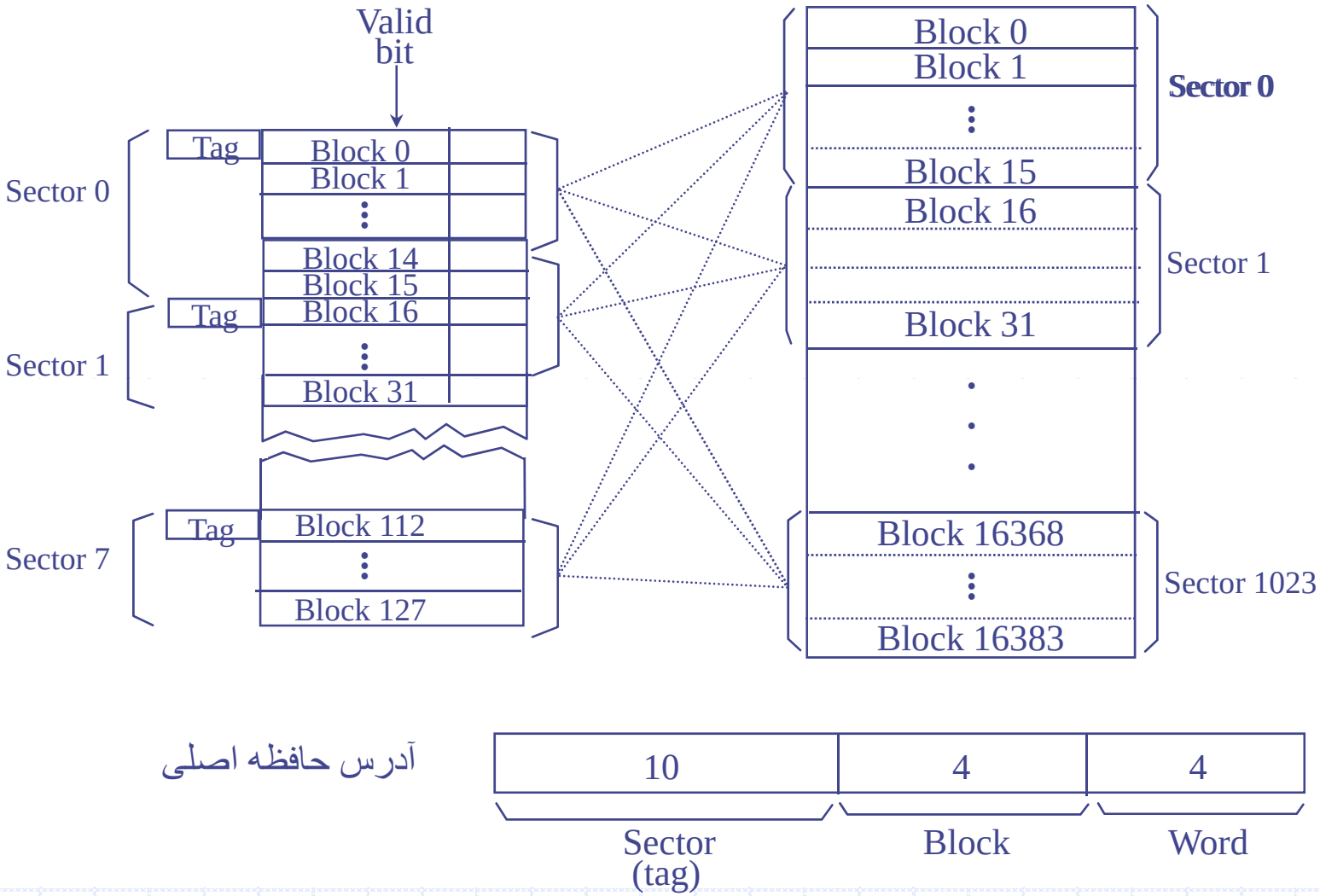
از این رو به هم پیوستن های 8 بیتی با
علامتهای 2 بیتی نیاز است

گونه های دیگر

- چندین کلمه=1بلاک



- Sector (IBM 360/85) - 16 sector x 16 block/sector
  - 1 سکتور:چندین بلاک متوالی
  - Cache miss:جایگزینی سکتور
  - بیت صحیح-یک بلاک بنا به درخواست انتقال داده شده



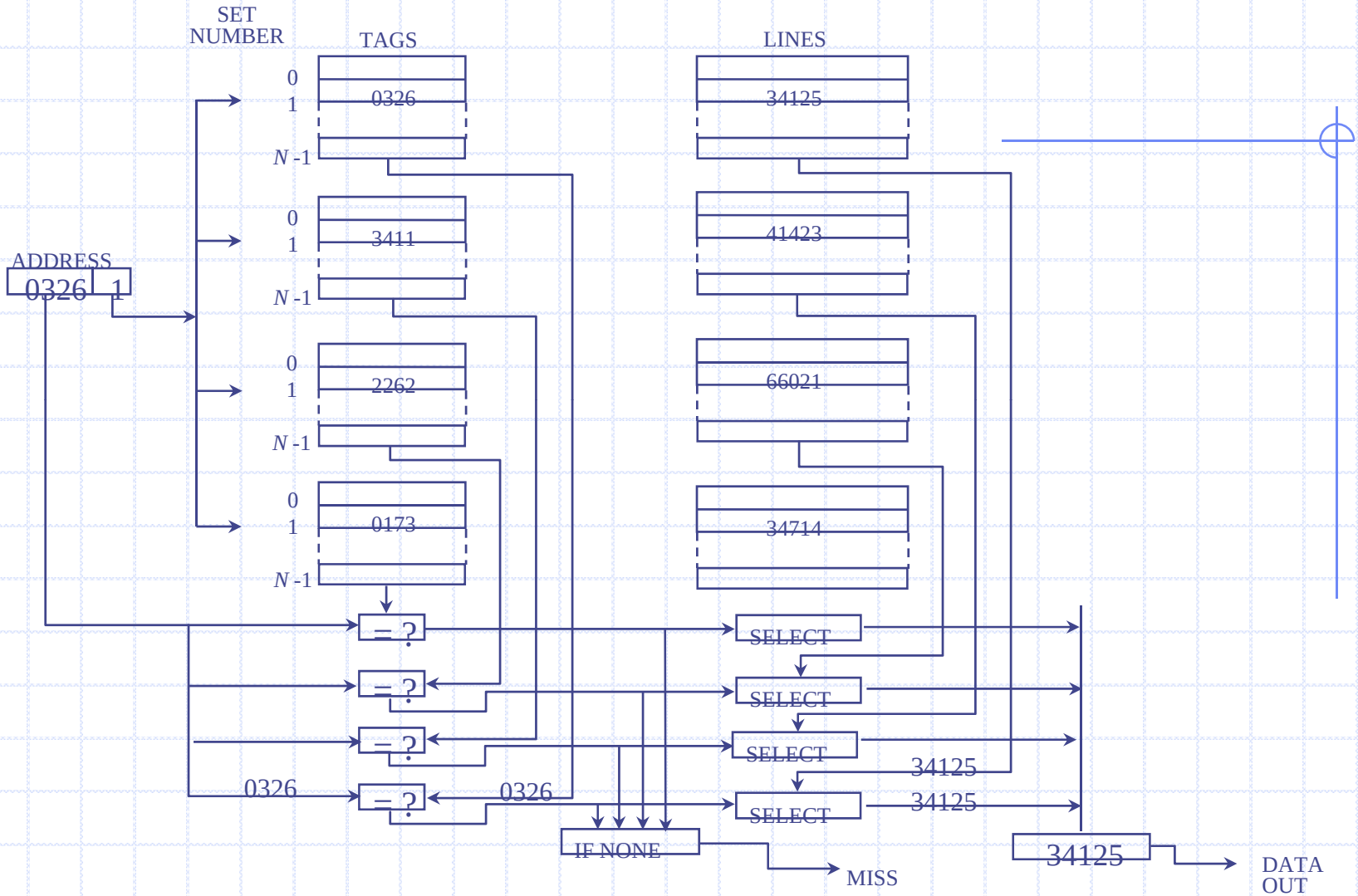
نگاشت سکتورcache

cont'd

Cache دارد $\frac{128 \text{ blocks}}{16 \text{ blocks/sector}}$ = **8** sector

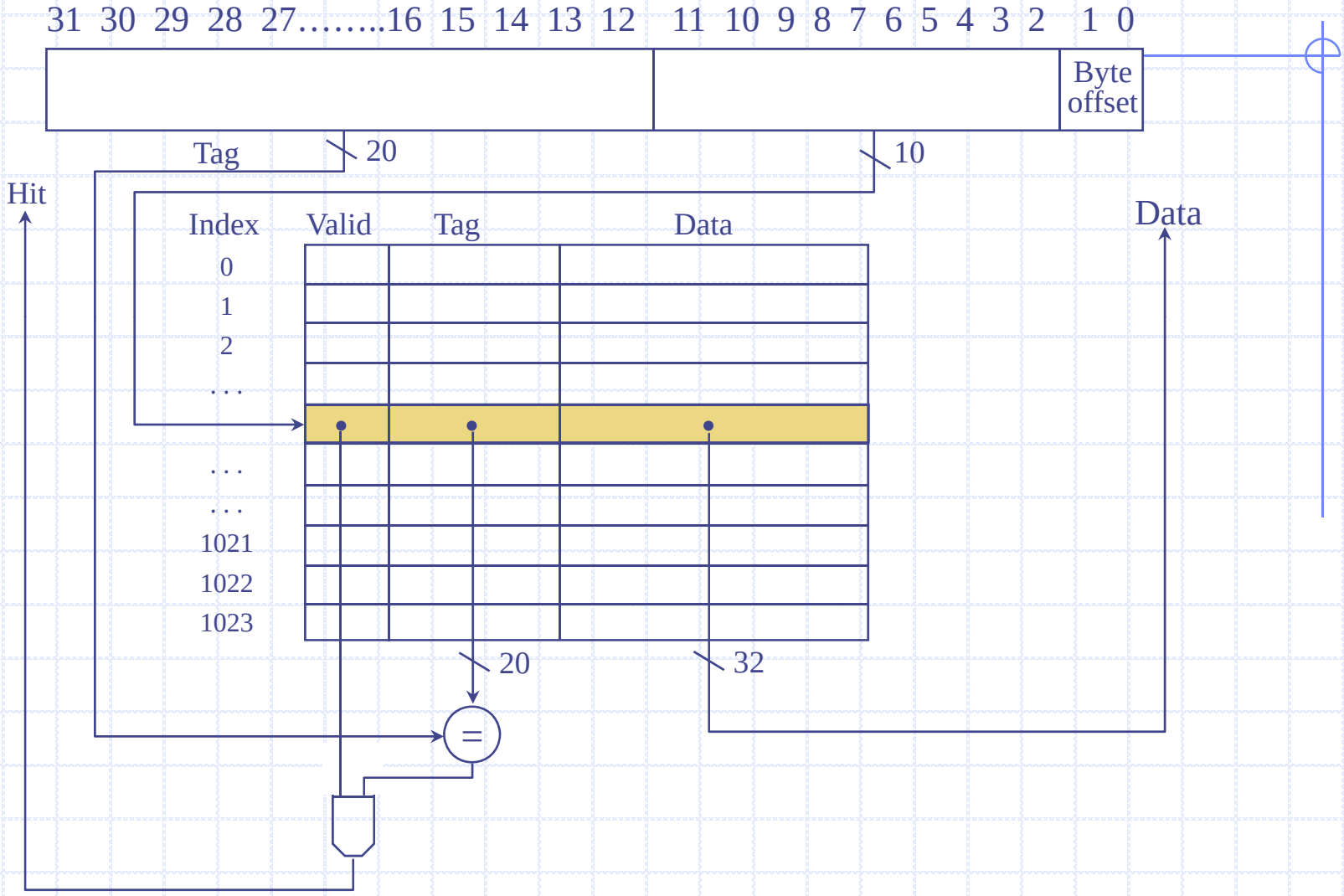
حافظه اصلی دارد ⇨ $\frac{16^k}{16}$ = **1K** sectors

cache نگاشت سکتور



The structure of a four-way set-associative cache with *N* sets

Address (showing bit positions)



جمع تمام بیت ها در cache

جمع بیتها =

(# of bits of a tag +

# of bits of a block +

# of bits in valid field) x Cache size

مثالی برای mips :

= ((32-14-2) + 32 + 1) x 2^{14}

= 2^{14} x 49

= 784 k bits

~ 100 kbytes

= 64 K (bytes)

= 2^{14} blocks with

فرض کنید یک نگاشت مستقیم cache.

به روز کردن حافظه اصلی/خط و مشي واکشی

- به روز کردنwrites are 8~16% :
  - باز نویسی
    - یک نوشتن اشتباه شاید ربطی به حافظه اصلی نداشته باشد
  - write-through
    - عبور و مرور بیشتر
- واکشی

چه طور خواندن/نوشتن رابه کار ببریم

- One-word-line: (DEC 3100)
- **خواندن:آسان است**
  - آدرس را به cache مناسب بفرستیدآ درس را از یکی از این دو pc(برای خواندن یک دستورالعمل) یا از ALU(برای دستیابی داده ها)
  - اگر سیگنال های Cache موفق شوند ،کلمه در خواست شده در خطوط داده ها موجود است.اگر که سیگنال های Cacheدچار مشکل شوند، ما آدرس کامل را به حافظه اصلی می فرستیم. زمانی که حافظه داده ها را بر می گرداند،ما آن را در cache می نویسیم.

- Read Miss is easy to be handled quickly:
  - خواندن تگ و خواندن بلاک می تواند باهم انجام شود
- نوشتن معمولاً کندتراست
  - خواندن تگ و نوشتن بلاک نمی تواند باهم انجام شود

(به جز برای one-word-line caches)

برای خط های چند کلمه ای:

<u>موقع نوشتن</u>

on a write miss, it is a

چرخه نوشتن - تغییردادن - خواندن

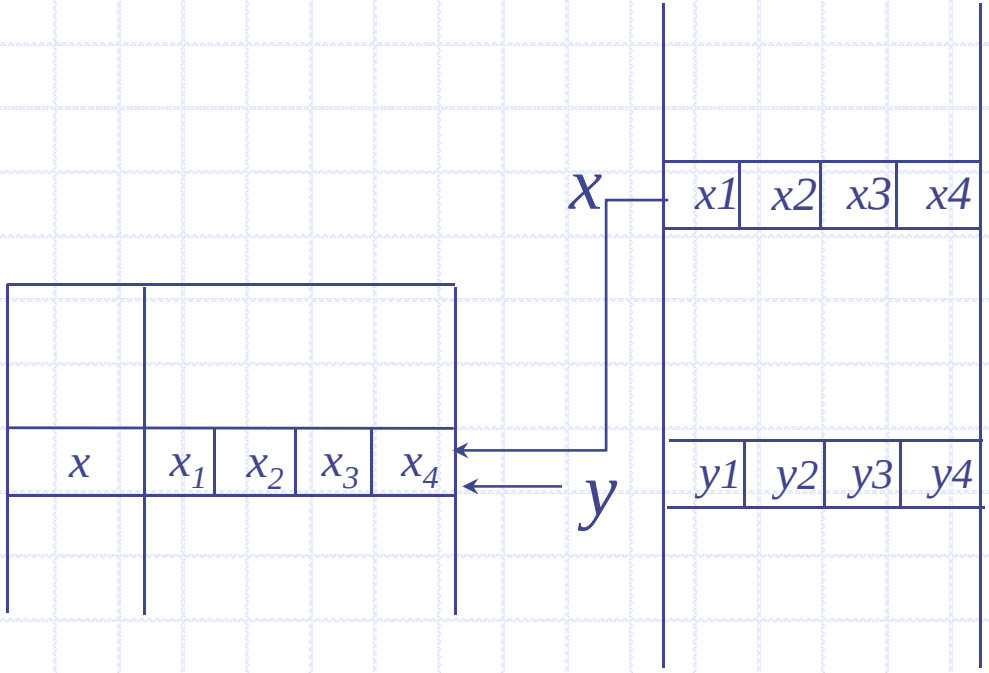
نوشتن بلاک

یک بخش

بلاک اصلی

مقایسه و سنجش تگ نمی تواند به موازات آن انجام گیرد در نتیجه انجام آن آهسته تر است

چرا؟ فرض کنید x,y در یک ستون یکسان نگاشت شوند و در ابتدای cache x را داریم



فرض کنید قبل از نوشتن ،cacheدر برگیرنده سطور x است،زمانی که y نوشته می شود یک فقدان ، و یک نوشته رخ می دهد

نوشتن y.3 z

Note after write miss, if not careful, we get

y	*x1*	*x2*	*Z*	*x4*

اما در پایان یک بازنویسی Y1,Y2,Y4نابود می شوند

- *Write through* (or *store through*) – اطلاعاتی که نوشته می شود روی هر دو بلاک cache و سطح پایین تر حافظه
- *Write back* (also called *copy back*) – اطلاعاتی که تنها روی بلاک در cache نوشته می شود. قالب cache تغییر یافته روی حافظه اصلی تنها زمانی که جایگزین شود نوشته می شود

باز نویسی

پاک کردن و پر کردن بلاک

یک بیت پر نشان می دهد که آیا یک سطر تا زمانی که در cache است تغییر کرده است .**موقعی** که یک "dirty line" جایگزین شود، باید در حافظه اصلی بازنویسی شود .

دو تا گزینه در write missوجود دارند

- *تخصیص نوشتن* (که به آن واکشی به منظور نوشتن هم گفته می شود)بلاک بار می شود ،به دنبال آن by the write-hit actions above. This is similar to a read miss.

- *No write allocate* (also called *write around*) - The block is modified in the lower level and not loaded into the cache.

Think what you do when have a write miss!

چه موقع write throughبهتر است؟

- زمانی که از cache دو سطحی استفاده می شود

Cpu روی یک تراشه cache کوچک است.

+

یک تراشه بزرگ cache

- سازگاری آسانتر است
- به وسیله دومین cacheاز ترافیک حافظه جلوگیری می شود

Write-back vs. Write-through

- **سرعت (باز نویسی سریع است)**
- **Traffic**(در حالت عمومی،باز کپی بهتر است)
  - If more than one read hit to the line
  - So attractive to multiprocessor in this sense
- **ساز گاری cache(write-through) بهتر است)**
- **منطقی**(copy-backبسیار کامل است)

Write-back vs. Write-through

- **Buffering(4 برای write-throughبهترین است)**

  به هردو نیاز دارد، در حالی که copy-backفقط به یکی نیاز دارد .

  مدیریت پیچیده است زیرا زمانی که یک ref ساخته می شود ،باید با buffer همراه باشد

- **قابلیت اطمینان(write-throughبهتر است)** برای اینکه حافظه اصلی قابلیت شناسایی خطا را دارد

  “انتخاب واضحی وجود ندارد “ برحسب کارایی

سایت مرجع دانشجوی پیام نور

- ✓ نمونه سوالات پیام نور : بیش از ۱۱۰ هزار نمونه سوال همراه با پاسخنامه تستی و تشریحی
- ✓ کتاب ، جزوه و خلاصه دروس
- ✓ برنامه امتحانات
- ✓ منابع و لیست دروس هر ترم
- ✓ دانلود کاملا رایگان بیش از ۱۴۰ هزار فایل مختص دانشجویان پیام نور