

# Definition of Hardware

The part of a system that can be kicked

# Basic Microarchitecture and Embedded processors in the market

# The Acronyms: CISC vs. RISC

**CISC** — **C**omplex **I**nstruction **S**et  
**C**omputers

**versus**

**RISC** — **R**educed **I**nstruction **S**et  
**C**omputers

# From CISC to RISC

- Why CISC?
  - Memory are **expensive** and **slow** back then
  - Cramming more functions into one instruction
  - Using microcode ROM ( $\mu$ ROM) for “complex” operations
- Justification for RISC
  - Complex apps are mostly composed of simple assignments
  - RAM speed catching up
  - Compiler (human) getting smarter
  - Frequency $\uparrow$   $\Rightarrow$  shorter pipe stages

<b>CISC</b>	<b>RISC</b>
Variable length instructions	Fixed-length instructions
Abundant instructions and addressing modes	Fewer instructions and addressing modes
Longer decoding	Easier decoding
Mem-to-mem operations	Load/store architecture
Use on-core microcode	No microinstructions, directly executed by HW logic
Less pipelineability	Better pipelineability
Closer semantic gap (shift complexity to microcode)	Needs smart compilers
IBM 360, DEC VAX, Intel IA32, Mot 68030	IBM 801, IBM RS6000, MIPS, Sun Sparc

# CISC vs. RISC (1970s – 80s)

**CISC**

**RISC**

	<b>IBM 370/168</b>	<b>VAX 11/780</b>	<b>Xerox Dorado</b>	<b>IBM 801</b>	<b>Berkeley RISC1</b>	<b>Stanford MIPS</b>
<b>Year</b>	<b>1973</b>	<b>1978</b>	<b>1978</b>	<b>1980</b>	<b>1981</b>	<b>1983</b>
<b># instructions</b>	<b>208</b>	<b>303</b>	<b>270</b>	<b>120</b>	<b>39</b>	<b>55</b>
<b>Microcode</b>	<b>54KB</b>	<b>61KB</b>	<b>17KB</b>	<b>0</b>	<b>0</b>	<b>0</b>
<b>Instruction size</b>	<b>2 to 6 B</b>	<b>2 to 57 B</b>	<b>1 to 3 B</b>	<b>4B</b>	<b>4B</b>	<b>4B</b>
<b>Execution model</b>	<b>Reg-reg Reg-mem Mem-mem</b>	<b>Reg-reg Reg-mem Mem-mem</b>	<b>Stack</b>	<b>Reg-reg</b>	<b>Reg-reg</b>	<b>Reg-reg</b>

Source: Andy Tanenbaum's Structured Computer Organization

# RISC Views

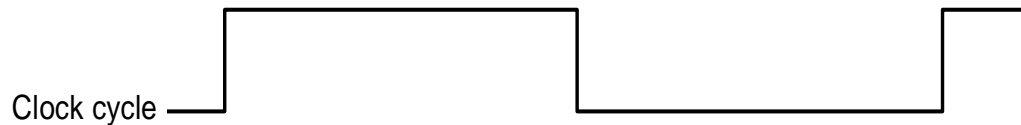
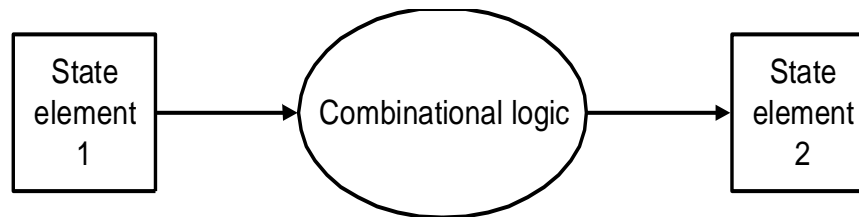
- From processor architect perspective
  - Simpler decoding and pipelining
  - More scalable
  - Interlock either provided by hardware or by compiler/scheduler
- From compiler perspective
  - A large “architecture register file” to manipulate
    - Due to free-up space from microcode ROM
    - How? e.g. Register windows or Register Stack Engine (RSE in Itanium)
  - Performance heavily rely on static code scheduling
    - Key: finding parallelism

# CISC Camp Strikes Back [Colwell et al. '85]

- "Instruction Set and Beyond: Computers, Complexity and Controversy" in IEEE Computer, Sept. 1985.
- RISC lessons are not incompatible or mutually exclusive
  - Large register file (register windows) is not a patent of RISCs
- Intel 432 study as an example
  - Decoding could be masked by execution
  - CISCy ISA plays no culprit of procedure call overheads
- Moore's Law will narrow the gap
- Compiler costs will dominate

# Our Implementation

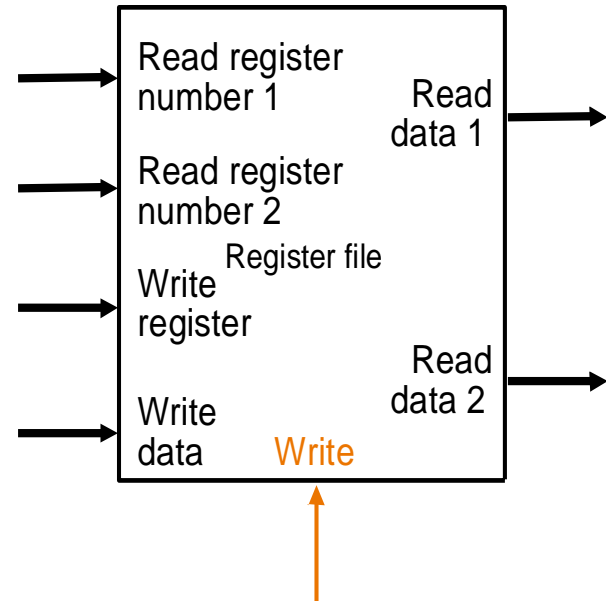
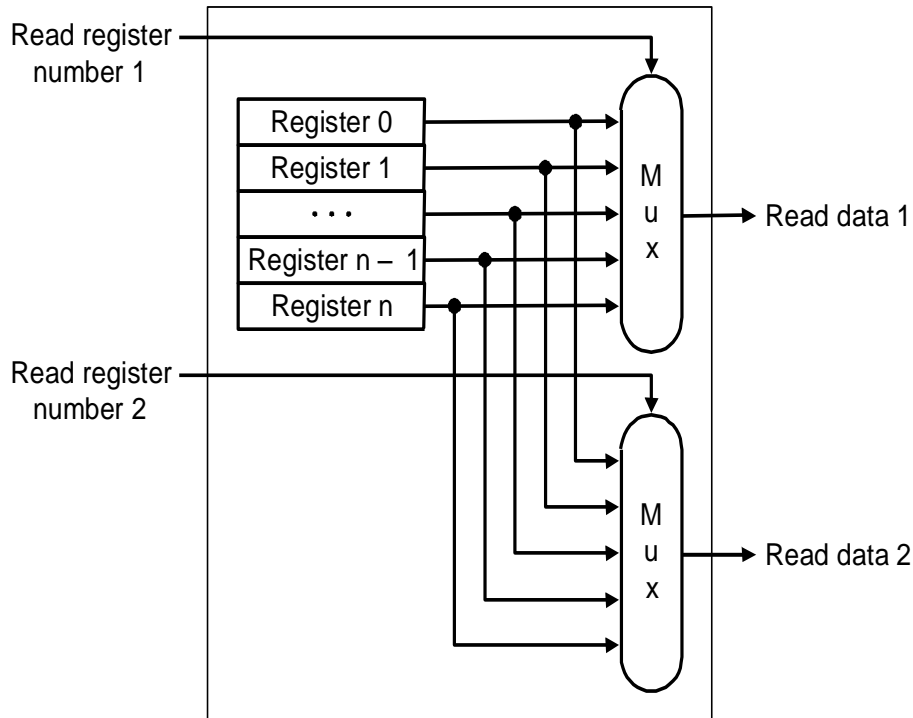
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements





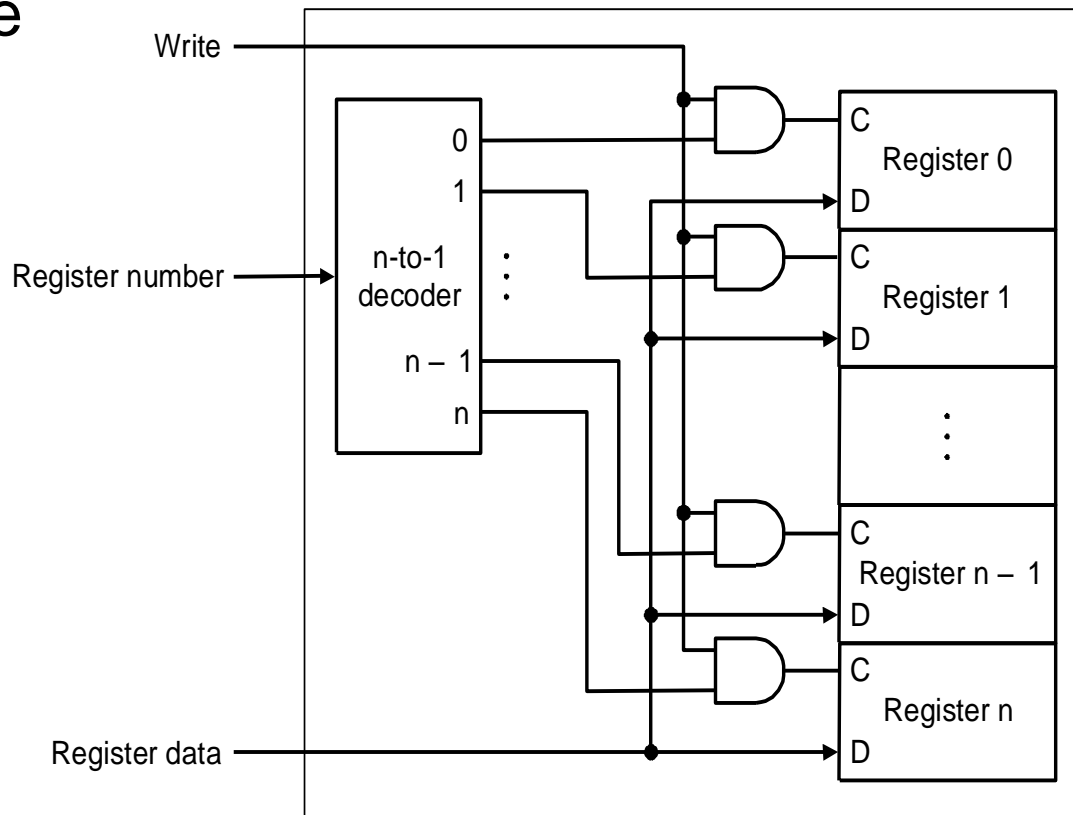
# Register File

- Built using D flip-flops



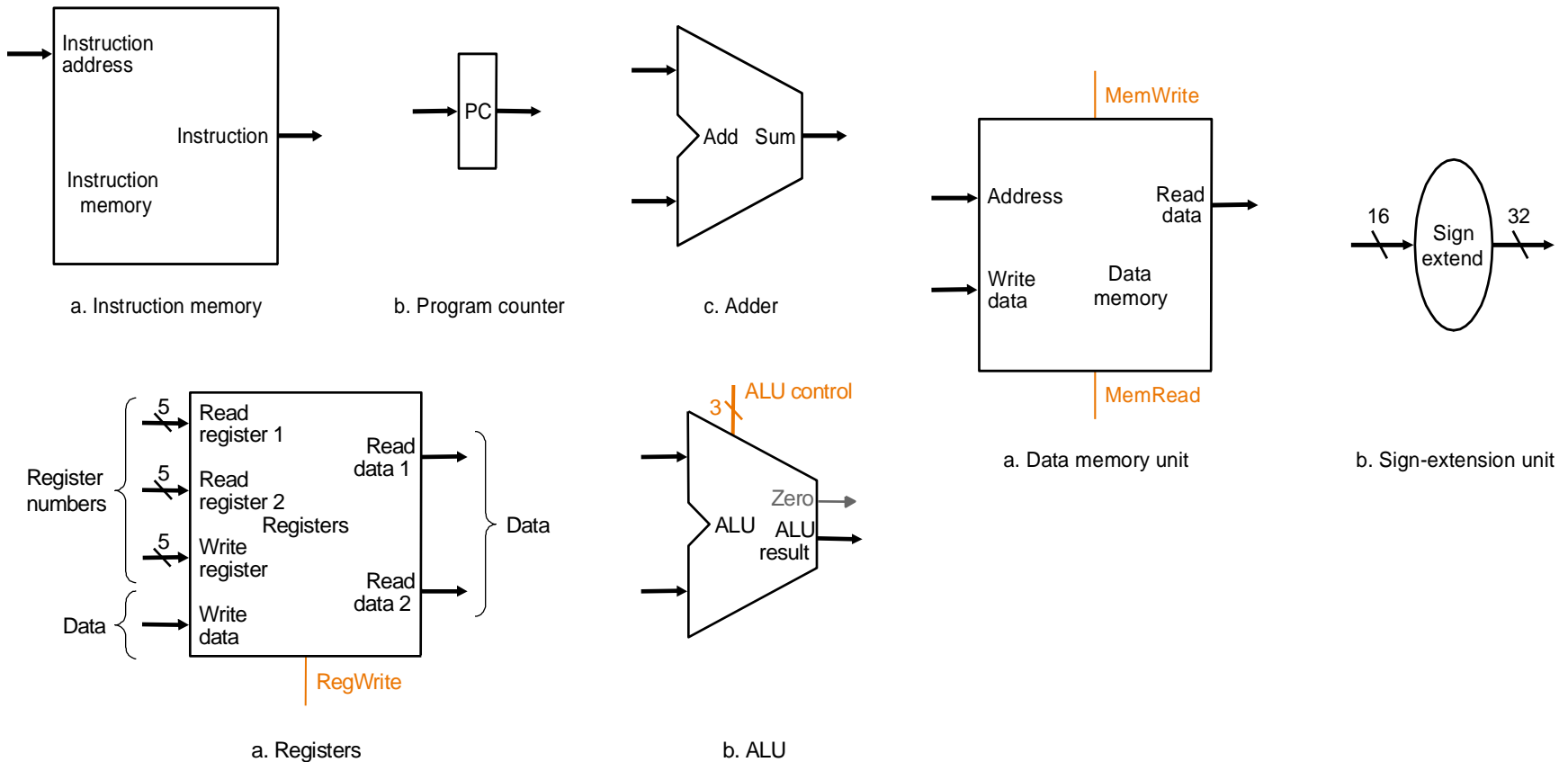
# Register File

- Note: we still use the real clock to determine when to write



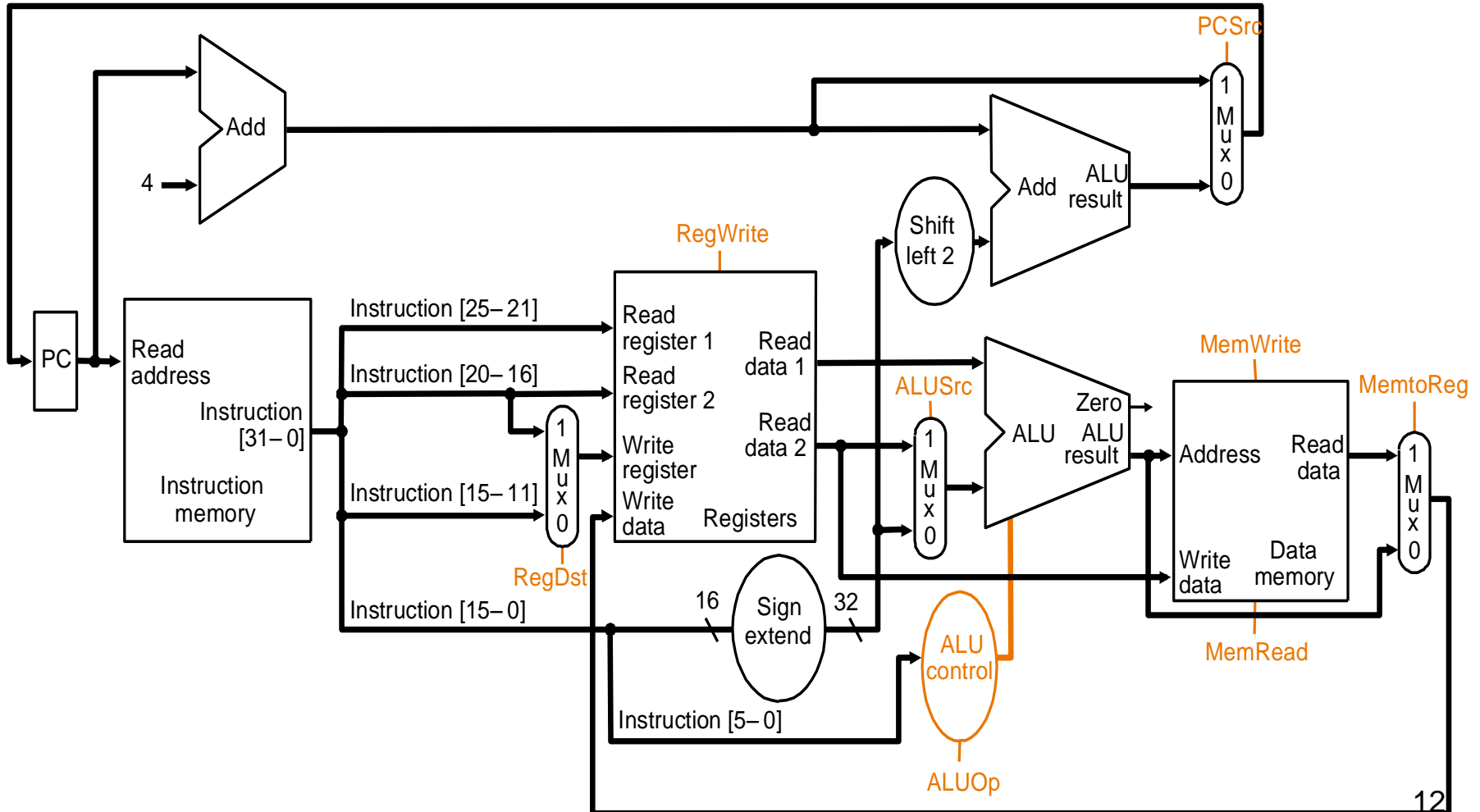
# Simple Implementation

- Include the functional units we need for each instruction

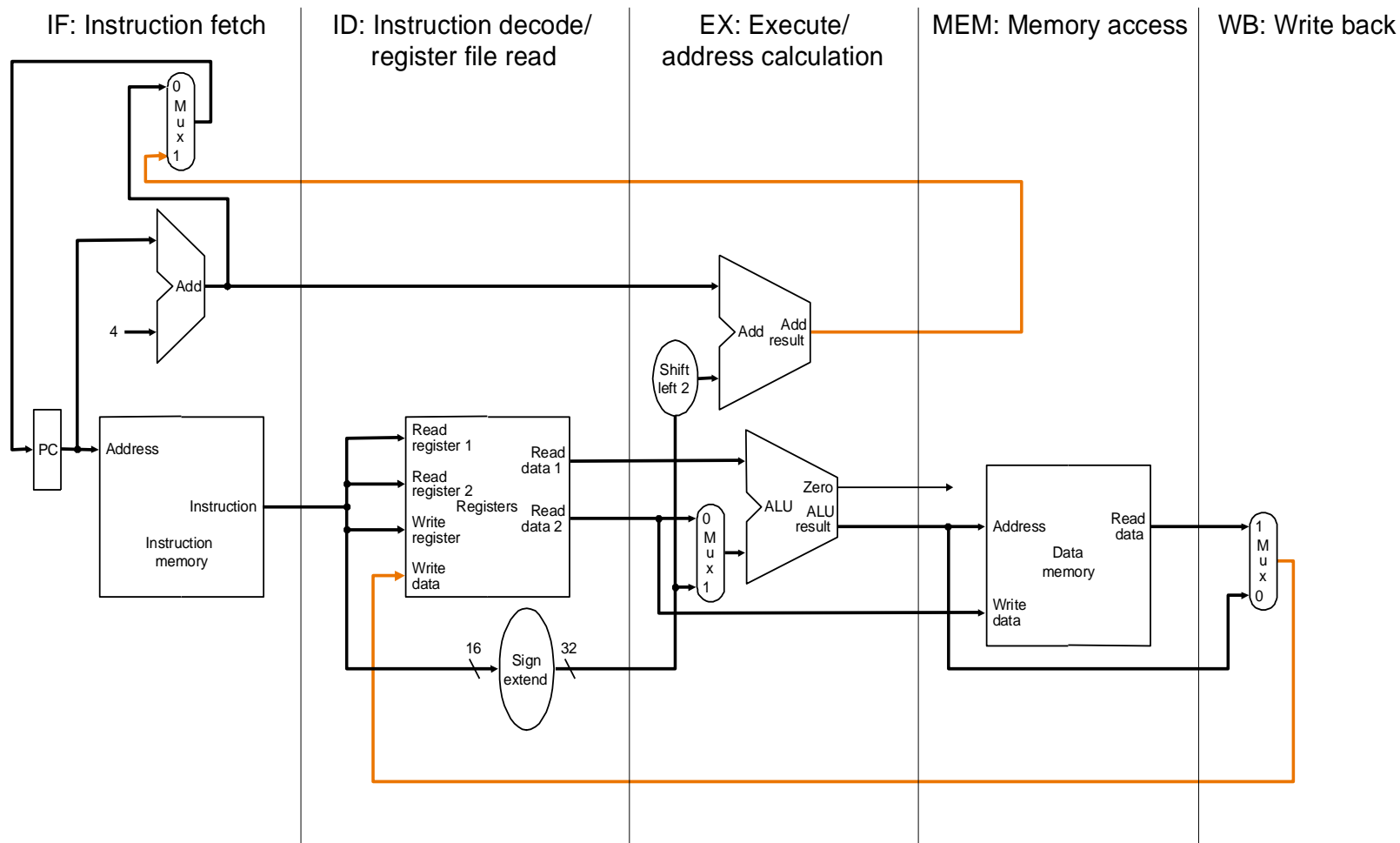


# Building the Datapath

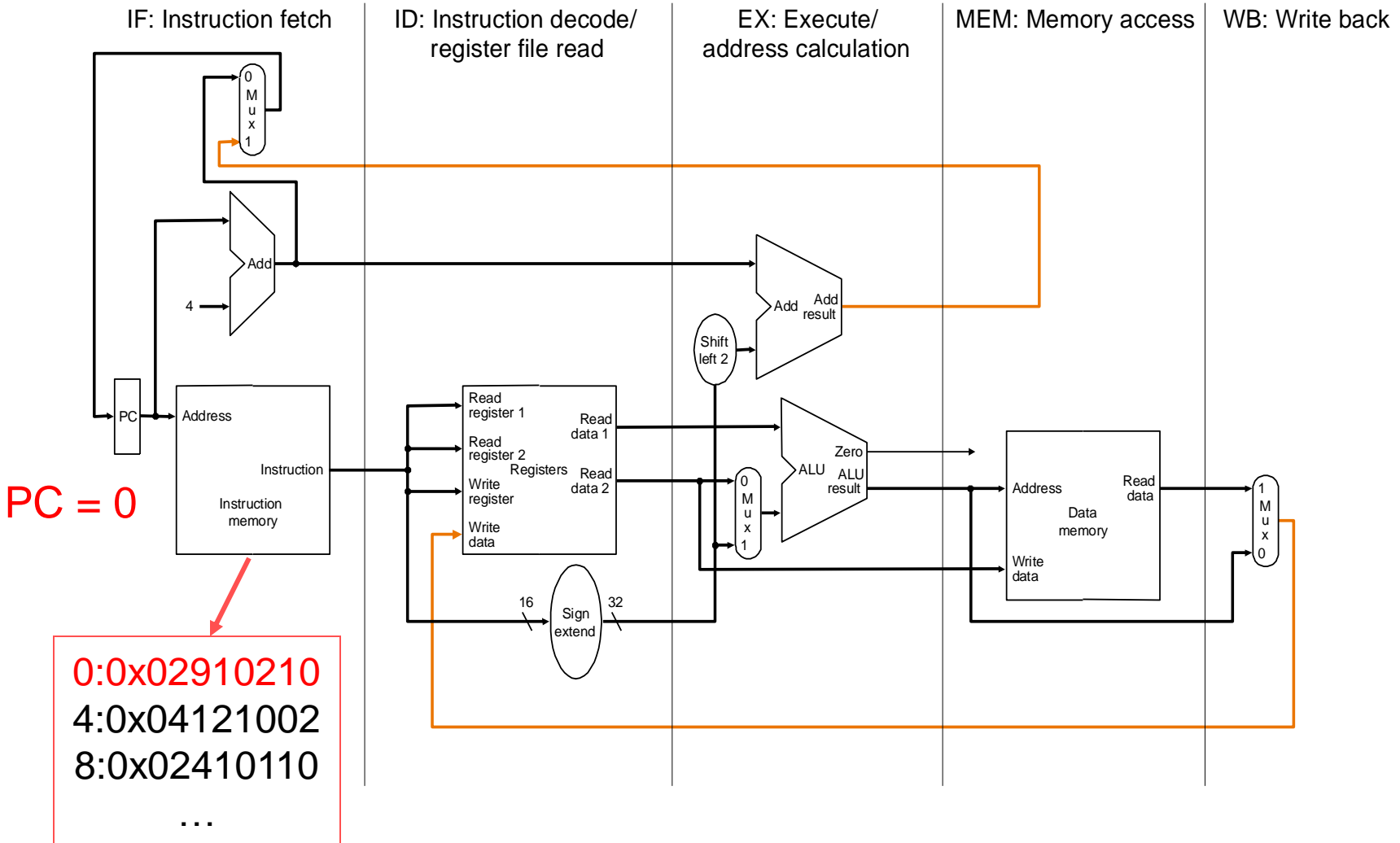
- Use multiplexers to stitch them together



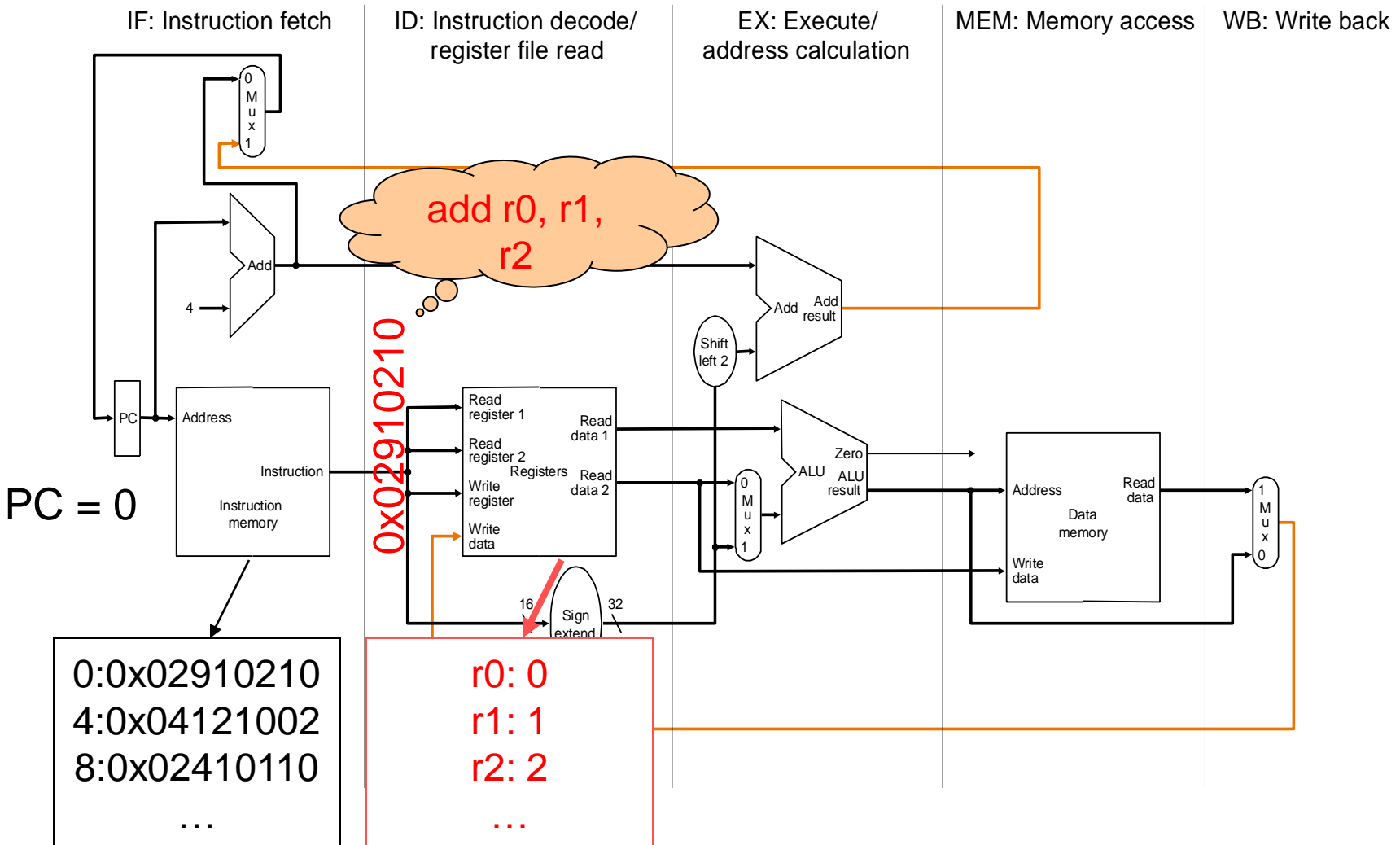
# Basic Architecture



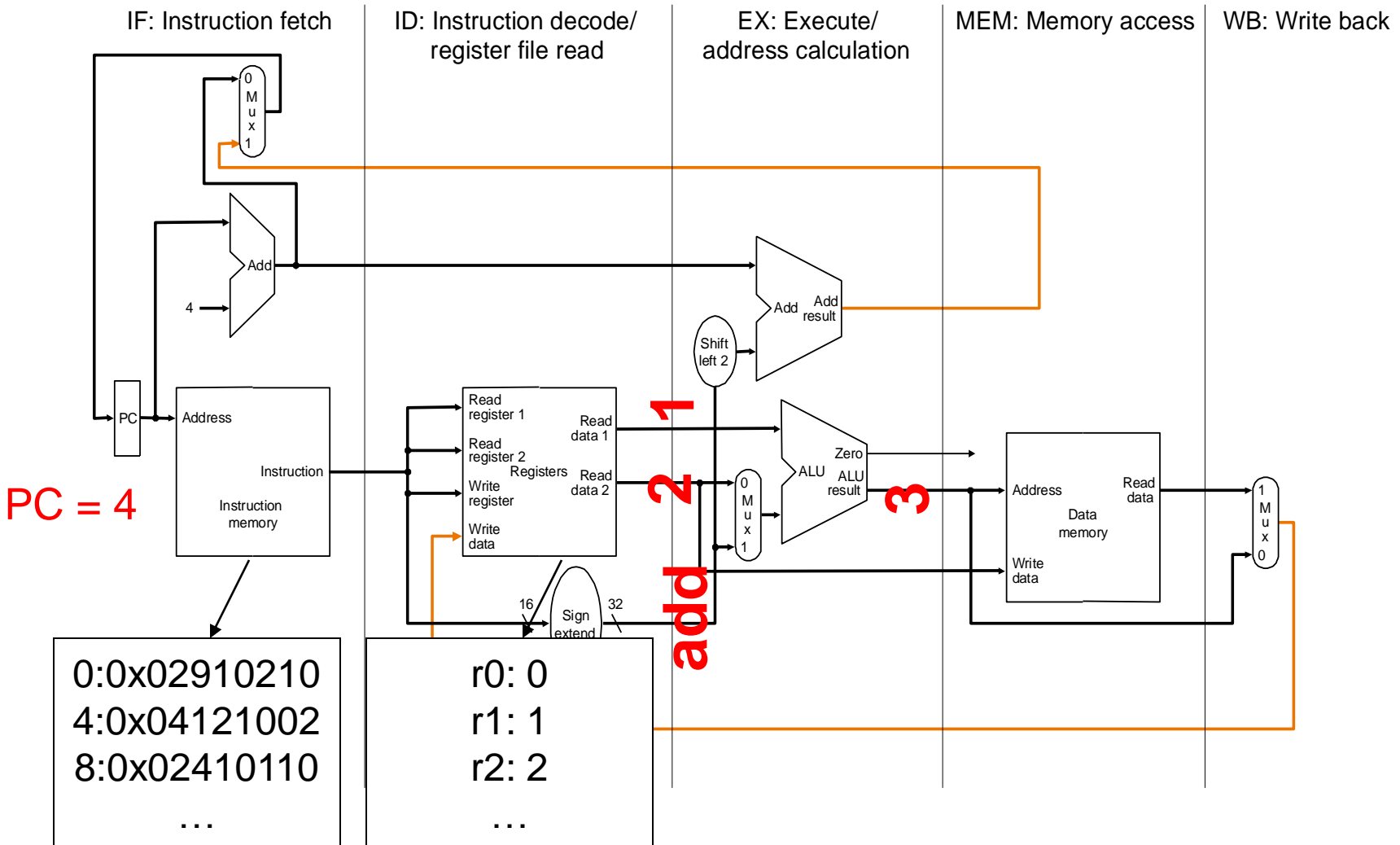
# Instruction Fetch



# Instruction Decoder

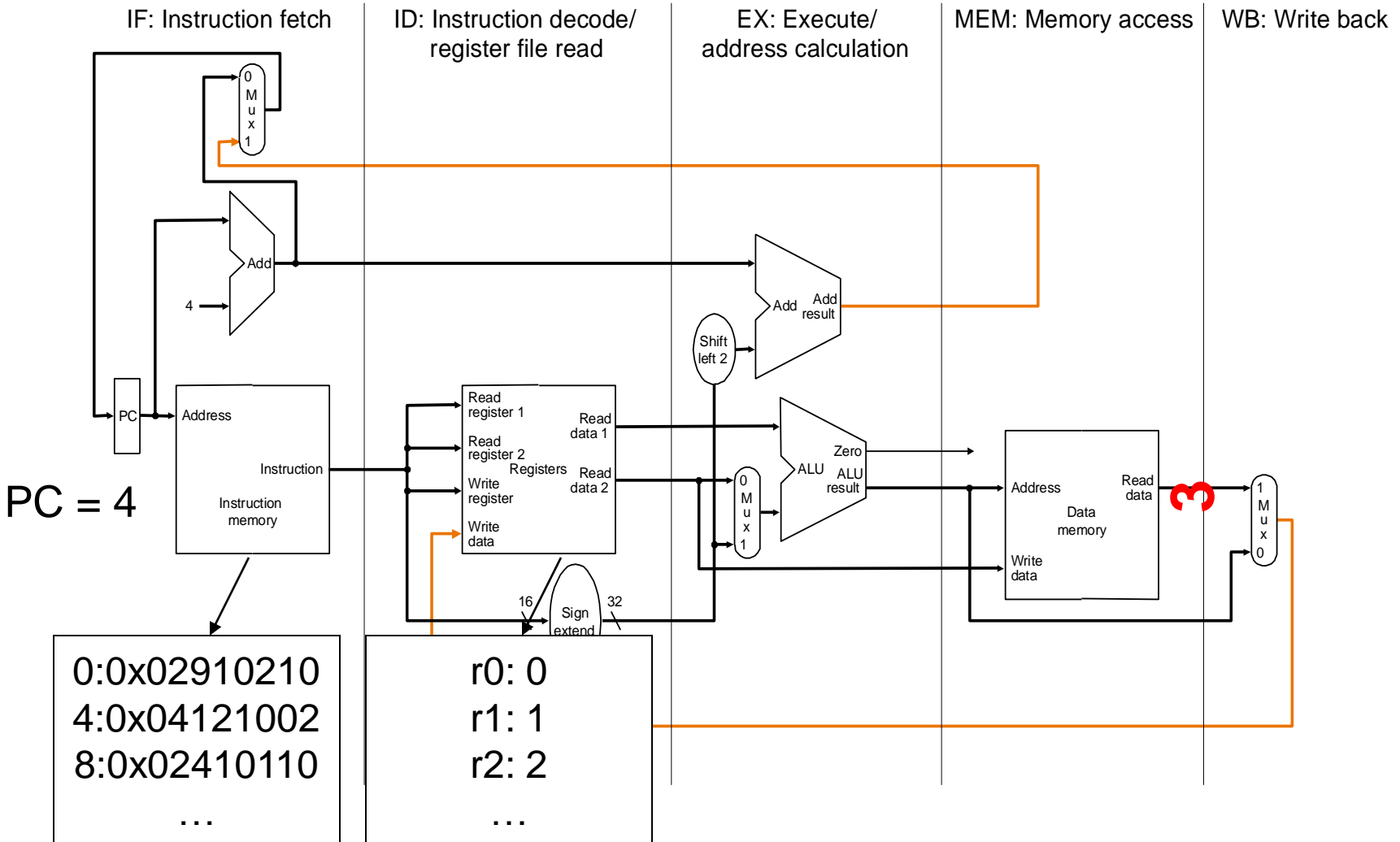


# Execution

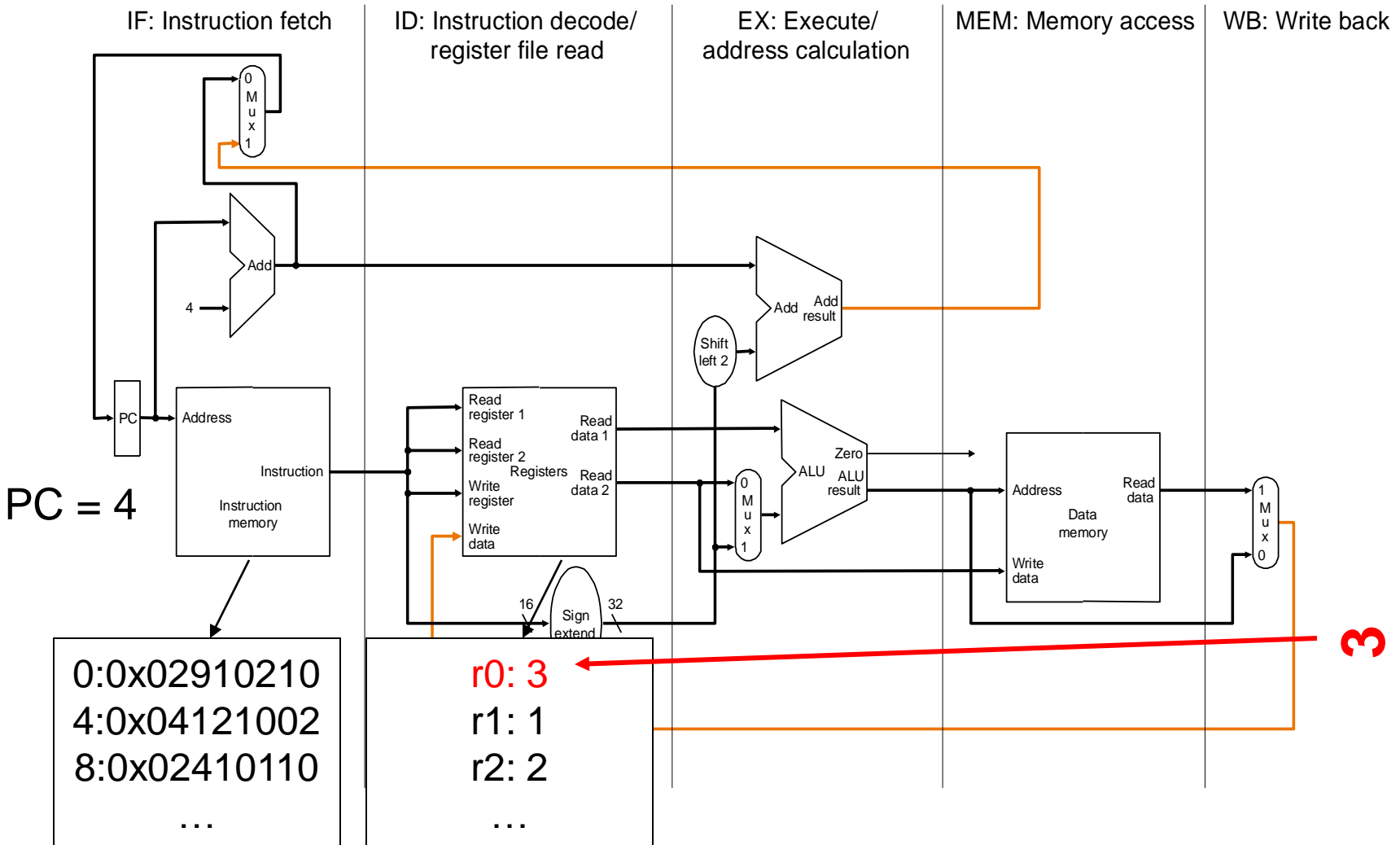




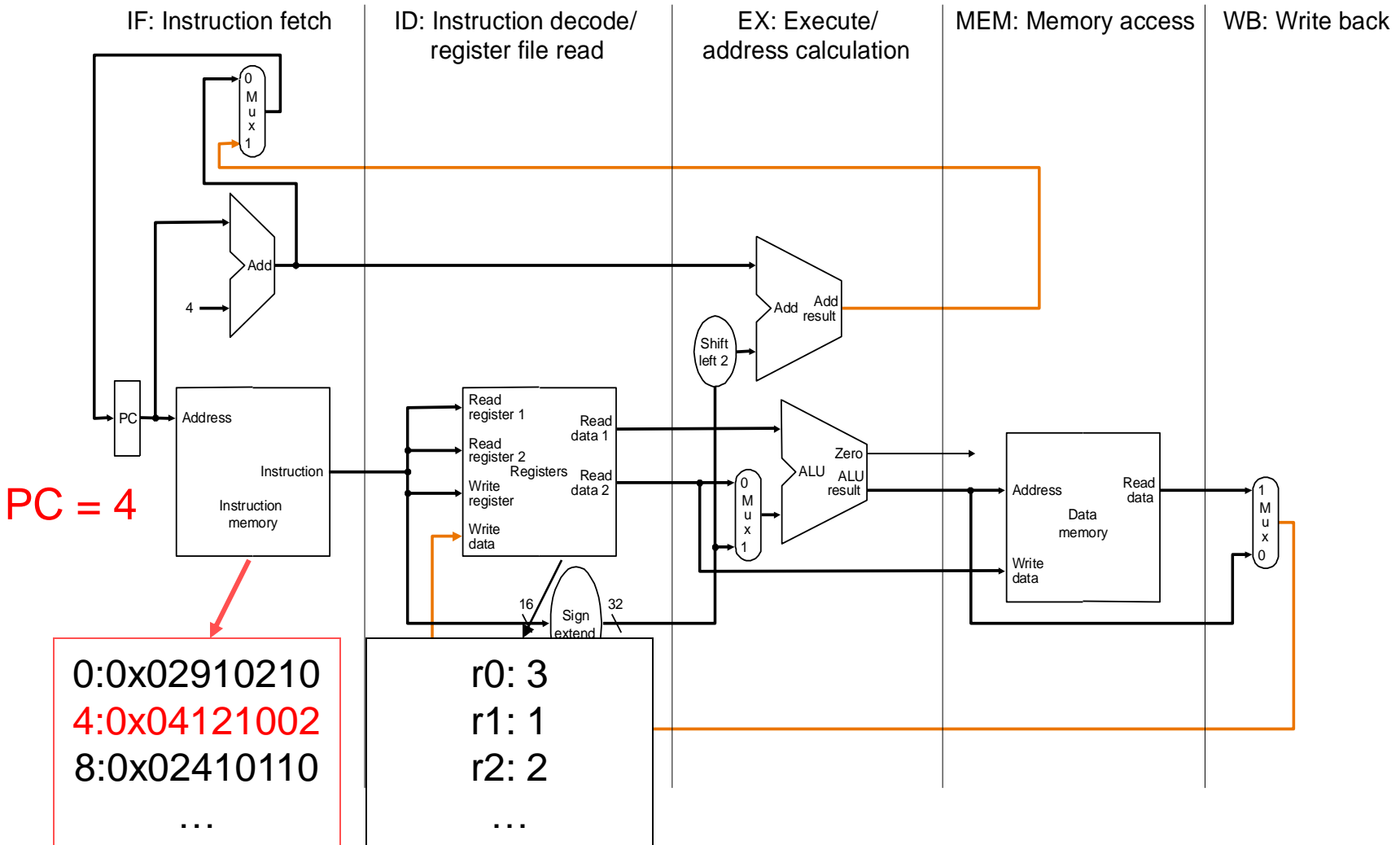
# Memory



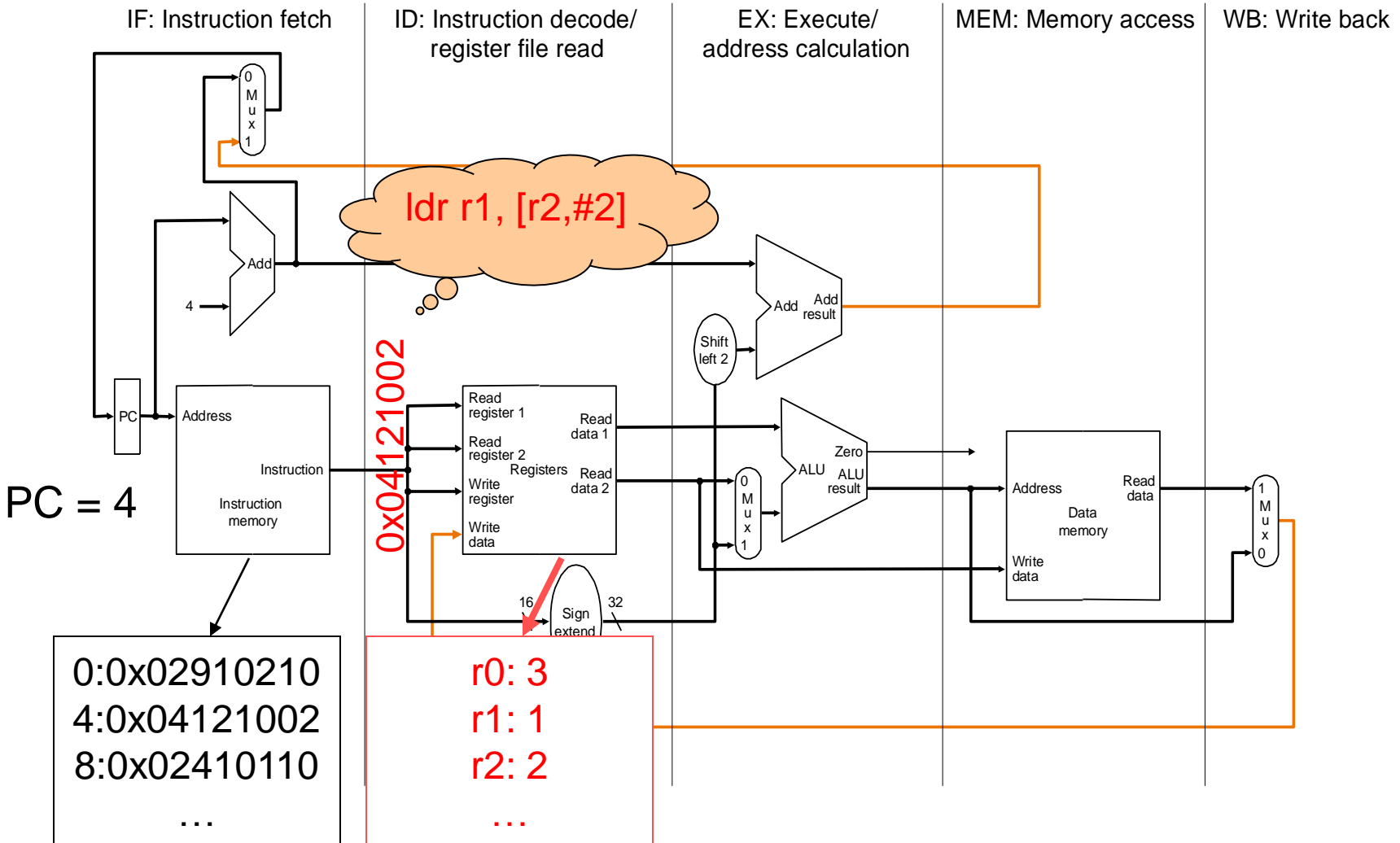
# Writeback



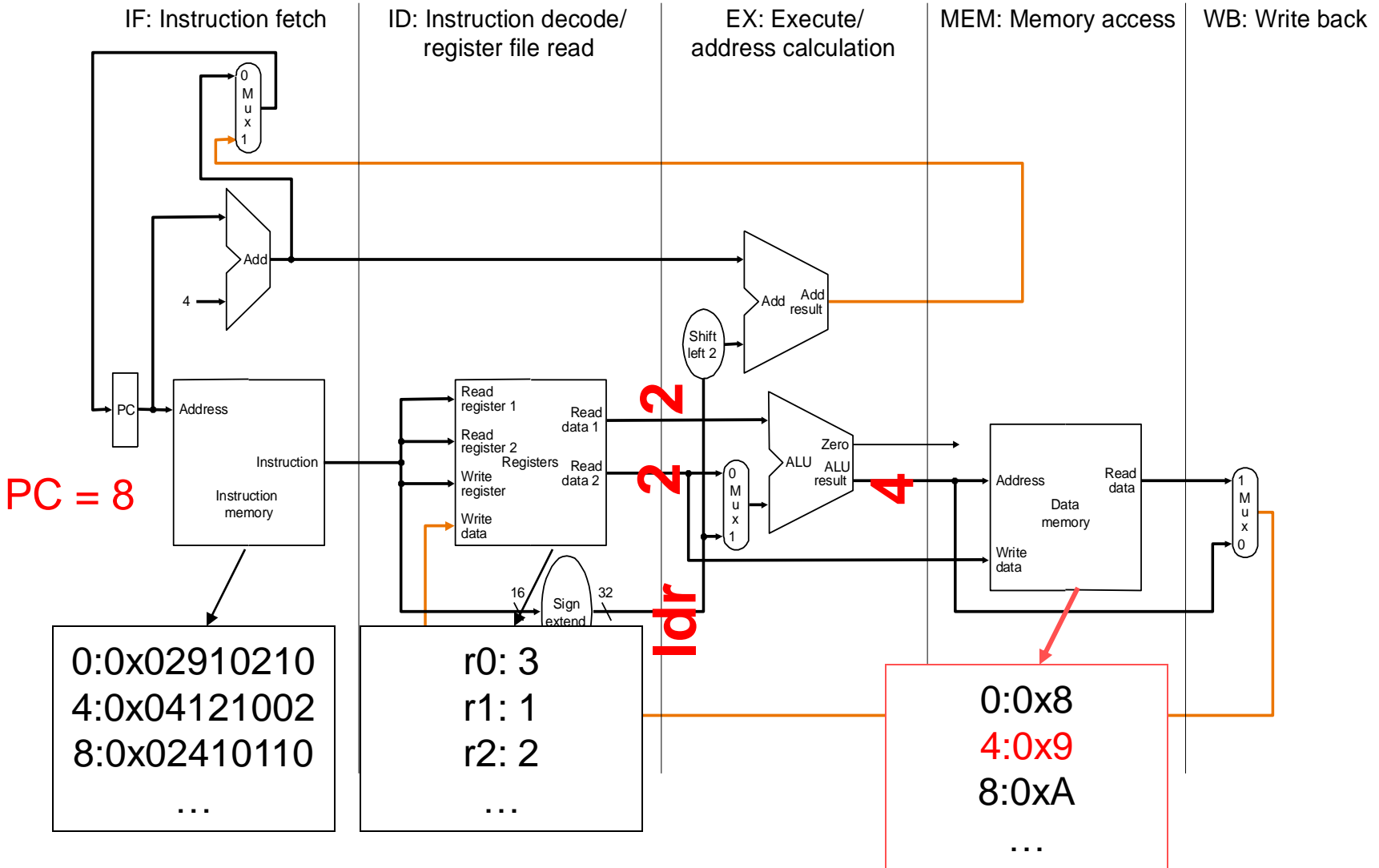
# Instruction Fetch



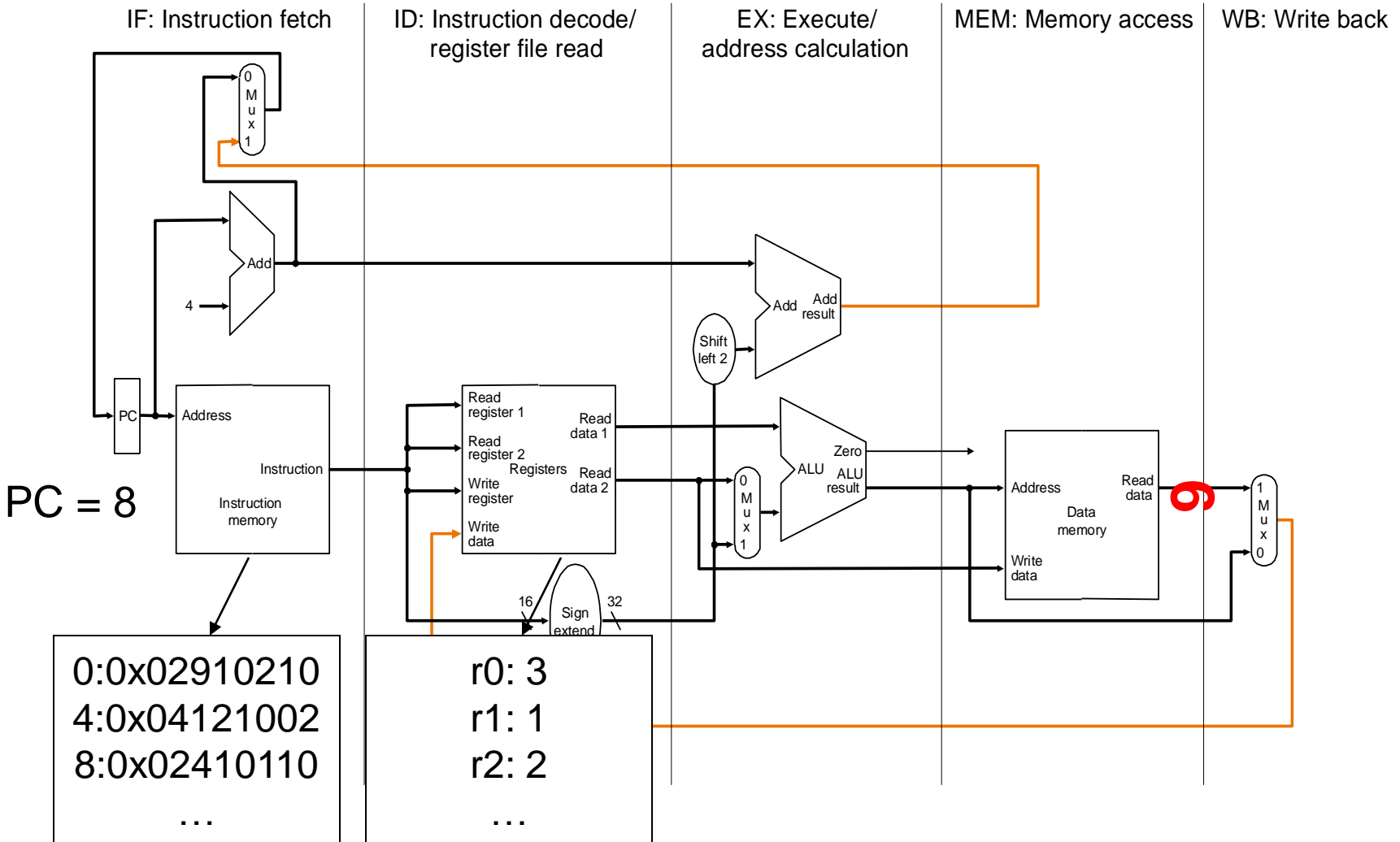
# Instruction Decoder



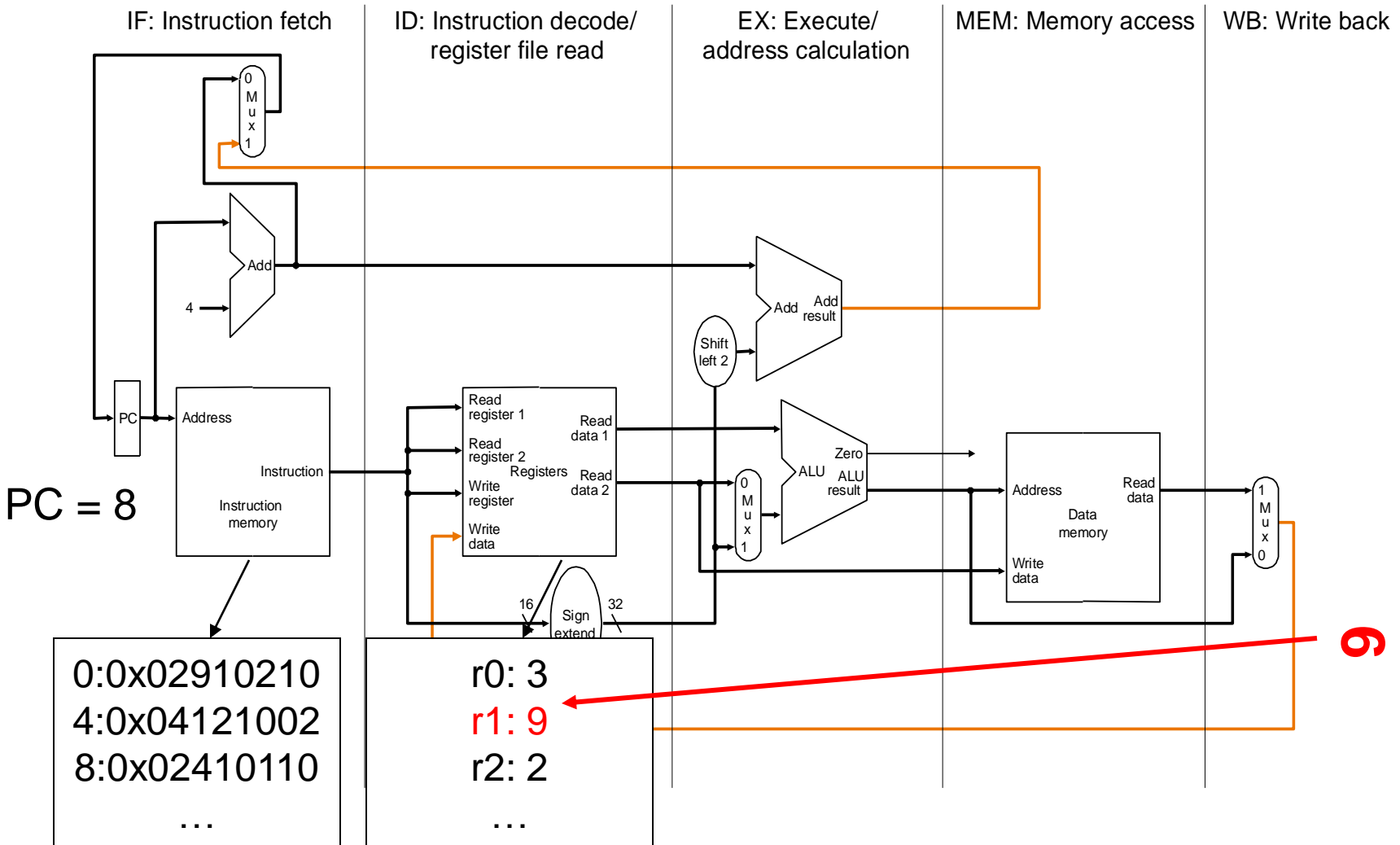
# Execution



# Memory



# Writeback



# Sequential Program Semantics



- Human expects “sequential semantics”
  - Tries to issue an instruction every clock cycle
  - There are dependencies, control hazards and long latency instructions
- To achieve performance with minimum effort
  - To issue more instructions every clock cycle
  - E.g., an embedded system can save power by exploiting instruction level parallelism and decrease clock frequency



# Pipelining

- What should be the steps to build a car washing machine?

# Car washing machine

Step 1: Rinse in fresh water mixed with soap

Step 2: Wash by scrubbing the car

Step 3: Blast in water jet

Step 4: Rinse in fresh water

Step 5: Wax

Step 6: Dry the car

# Car washing machine

- Each step is independent
- To wash more cars, we can do all steps simultaneously

Time	1	2	3	4	5	6	7	8	9	10
Car1	Step1	Step2	Step3	Step4	Step5	Step6				
Car2		Step1	Step2	Step3	Step4	Step5	Step6			
Car3			Step1	Step2	Step3	Step4	Step5	Step6		
Car4				Step1	Step2	Step3	Step4	Step5	Step6	
Car5					Step1	Step2	Step3	Step4	Step5	Step6
Car6						Step1	Step2	Step3	Step4	Step5
Car7							Step1	Step2	Step3	Step4
Car8								Step1	Step2	Step3
Car9									Step1	Step2
Car10										Step1

# Sequential processing

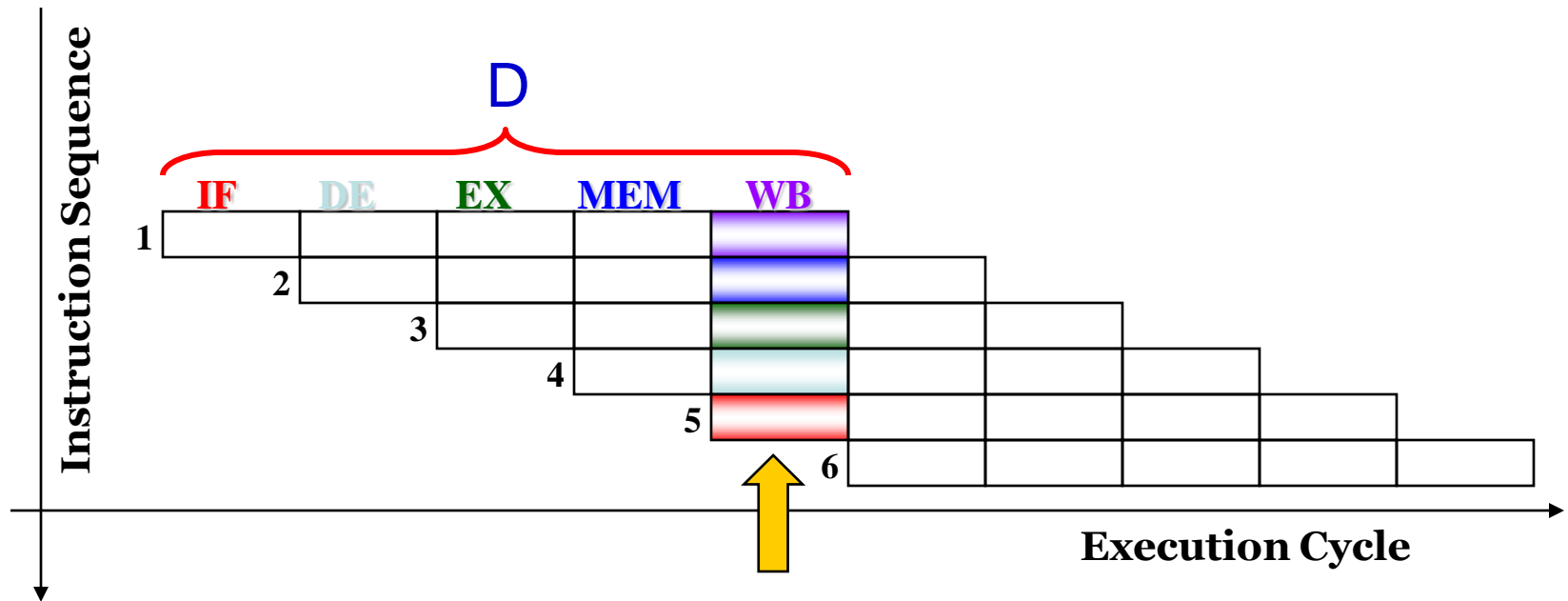
Cycle	1	2	3	4	5	6	7	8	9	10
Instr1	Fetch	Decode	Execute	Memory	Writeback					
Instr2						Fetch	Decode	Execute	Memory	Writeback

# Pipelining example

Cycle	1	2	3	4	5	6	7	8	9	10
Instr1	Fetch	Decode	Execute	Memory	Writeback					
Instr2		Fetch	Decode	Execute	Memory	Writeback				
Instr3			Fetch	Decode	Execute	Memory	Writeback			
Instr4				Fetch	Decode	Execute	Memory	Writeback		
Instr5					Fetch	Decode	Execute	Memory	Writeback	
Instr6						Fetch	Decode	Execute	Memory	Writeback
Instr7							Fetch	Decode	Execute	Memory
Instr8								Fetch	Decode	Execute
Instr9									Fetch	Decode
Instr10										Fetch

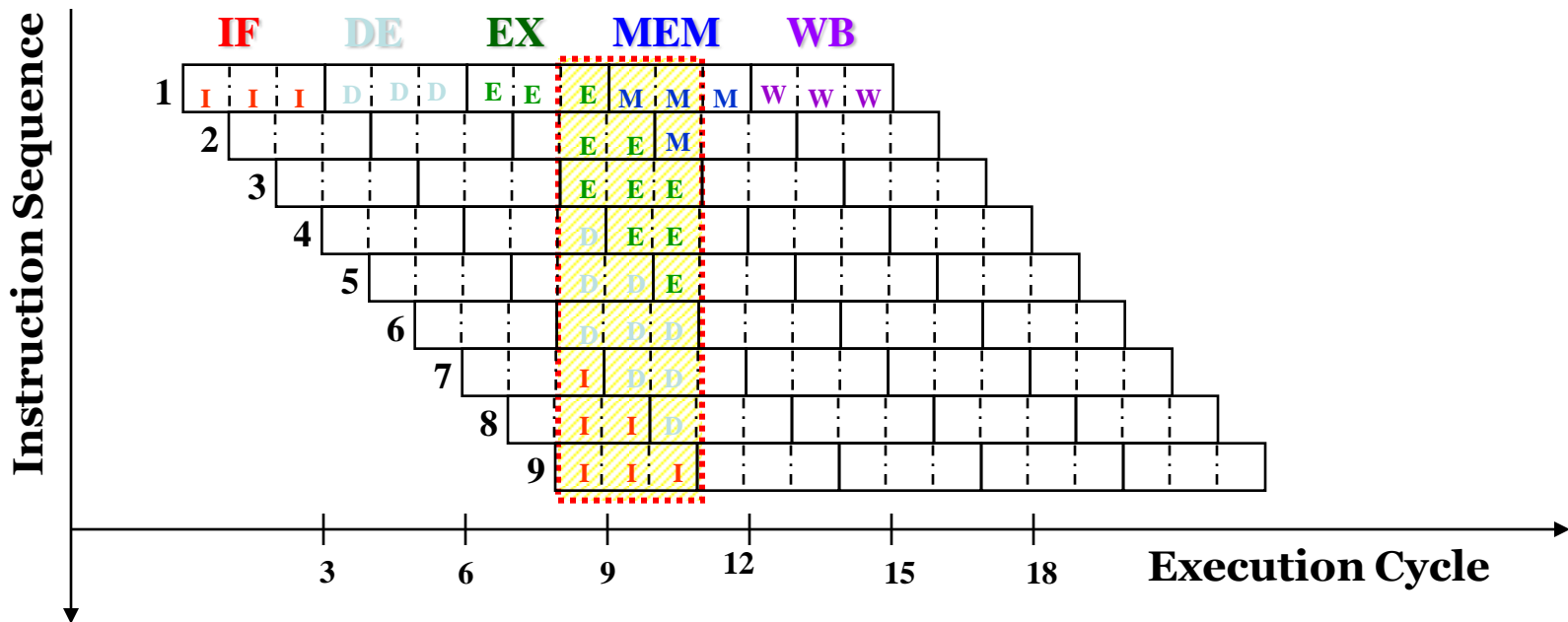
# Scalar Pipeline (Baseline)

- Machine Parallelism =  $D$  (= 5)
- Issue Latency (IL) = 1
- Peak IPC = 1, IPC = instruction per cycle =  $1/\text{CPI}$
- Pipeline depth = 5



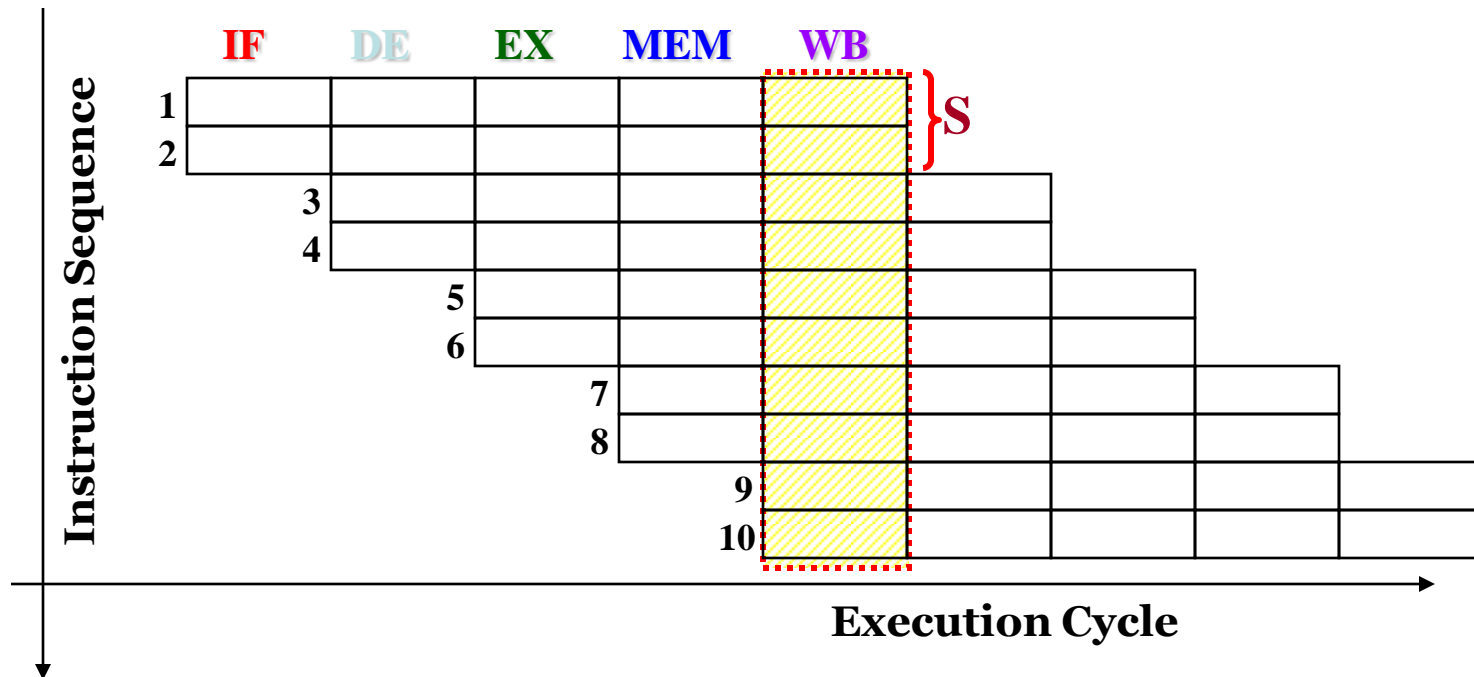
# Superpipelined Machine

- 1 major cycle = M minor cycles
- Machine Parallelism =  $M \times D$  (= 15) per major cycle
- Issue Latency (IL) = 1 minor cycles
- Peak IPC = 1 per minor cycle = M per baseline cycle
- Superpipelined machines are essentially deeper pipelined
- Pipeline depth = 15



# Superscalar Machine

- Can issue > 1 instruction per cycle by hardware
- Replicate resources, e.g. multiple adders or multi-ported data caches
- Machine Parallelism =  $S \times D$  ( $= 10$ ) where  $S$  is superscalar degree
- Issue Latency (IL) = 1
- IPC = 2





# But, an instruction is not a car!

- Control dependency
- Data dependency
- Resource dependency

# Instruction-Level Parallelism (ILP)

- Fine-grained parallelism
  - Multiple operations can be executed simultaneously
  - Programs' property
  - Independent on hardware technology improvements
- Enabled and improved by RISC
  - More ILP of a RISC over CISC does not imply a better overall performance
  - CISC can be implemented like a RISC
- A measure of inter-instruction dependency in an app
  - ILP assumes a unit-cycle operation, infinite resources, perfect frontend
  - $ILP \neq IPC$
  - $IPC = \# \text{ instructions} / \# \text{ cycles}$
  - ILP is the upper bound of attainable IPC
- Limited by
  - Data dependency
  - Control dependency

# ILP Example

- True dependency forces “sequentiality”
- ILP =  $3/3 = 1$
- False dependency removed
- ILP =  $3/2 = 1.5$

**c1=i1: load r2, (r12)**

**c2=i2: add r1, r2, 9**

**c3=i3: mul r2, r5, r6**

**i1: load r2, (r12)**

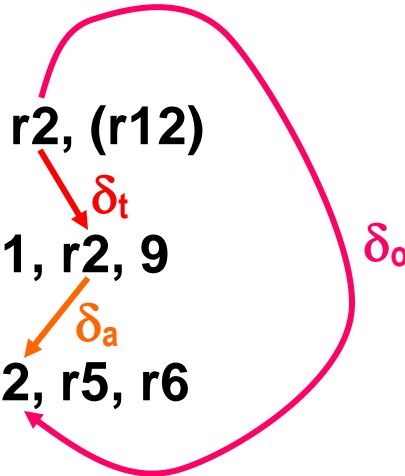
**i2: add r1, r2, 9**

**i3: mul r8, r5, r6**

**c1: load r2, (r12)**

**c2: add r1, r2, #9**

**mul r8, r5, r6**



# ILP, Another Example

When only 4 registers available

Load R1, 8(R0)

$R3 = R1 - 5$

$R2 = R1 * R3$

Store R2, 24(R0)

Load R1, 16(R0)

$R3 = R1 - 5$

$R2 = R1 * R3$

Store R2, 32(R0)

ILP =

# ILP, Another Example

When more registers (or register renaming) available

Load R1, 8(R0)

R3 = R1 - 5

R2 = R1 \* R3

Store R2, 24(R0)

Load R5, 16(R0)

R6 = R5 - 5

R7 = R5 \* R6

Store R7, 32(R0)

ILP =

# Window in Search of ILP

ILP = 1

$$R5 = 8(R6)$$

$$R7 = R5 - R4$$

$$R9 = R7 * R7$$

ILP = 1.5

$$R15 = 16(R6)$$

$$R17 = R15 - R14$$

$$R19 = R15 * R15$$

ILP = ?

# Window in Search of ILP

$$R5 = 8(R6)$$

$$R7 = R5 - R4$$

$$R9 = R7 * R7$$

$$R15 = 16(R6)$$

$$R17 = R15 - R14$$

$$R19 = R15 * R15$$

# Window in Search of ILP

$$R5 = 8(R6)$$

$$R15 = 16(R6)$$

$$R7 = R5 - R4$$

$$R17 = R15 - R14$$

$$R19 = R15 * R15$$

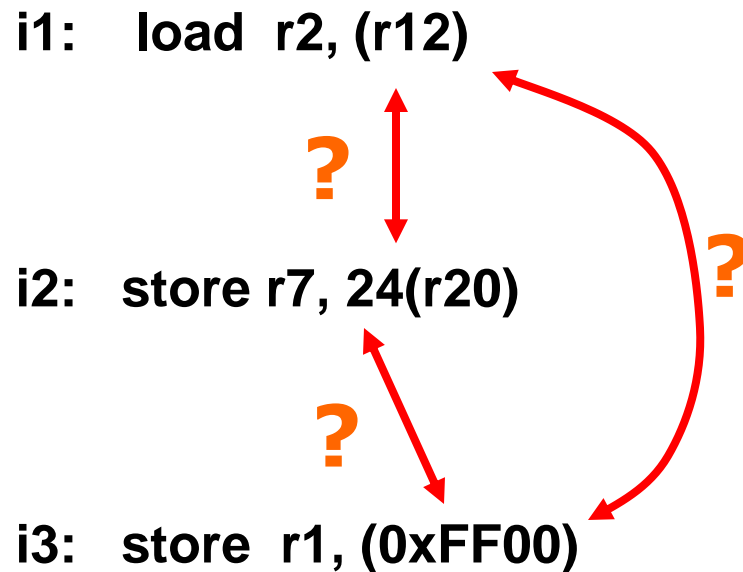
$$R9 = R7 * R7$$

- $ILP = 6/3 = 2$  better than 1 and 1.5
- Larger window gives more opportunities
- Who exploit the instruction window?
- But what limits the window?



# Memory Dependency

- Ambiguous dependency also forces “sequentiality”
- To increase ILP, needs dynamic memory disambiguation mechanisms that are either safe or recoverable
- ILP could be 1, could be 3, depending on the actual dependence



# Concept of Basic Blocks

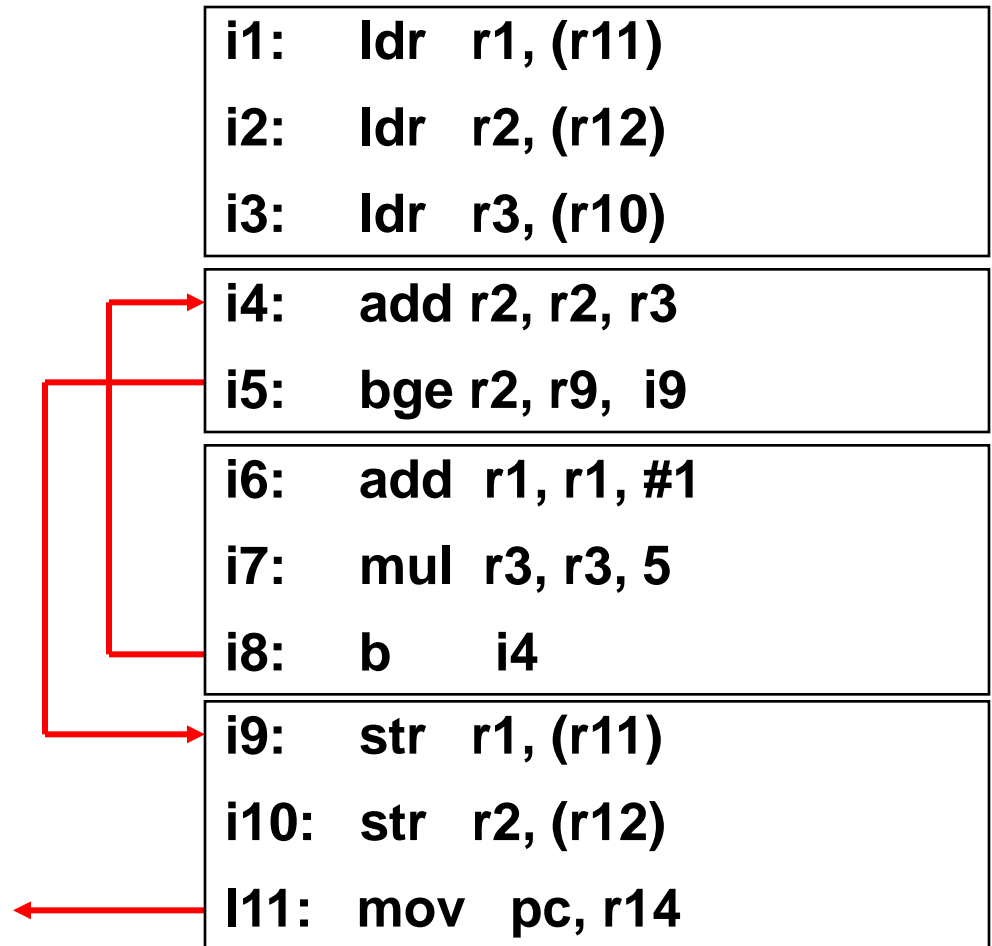
```
a = array[i];  
b = array[j];  
c = array[k];  
d = b + c;  
while (d<t) {  
    a++;  
    c *= 5;  
    d = b + c;  
}  
array[i] = a;  
array[j] = d;
```



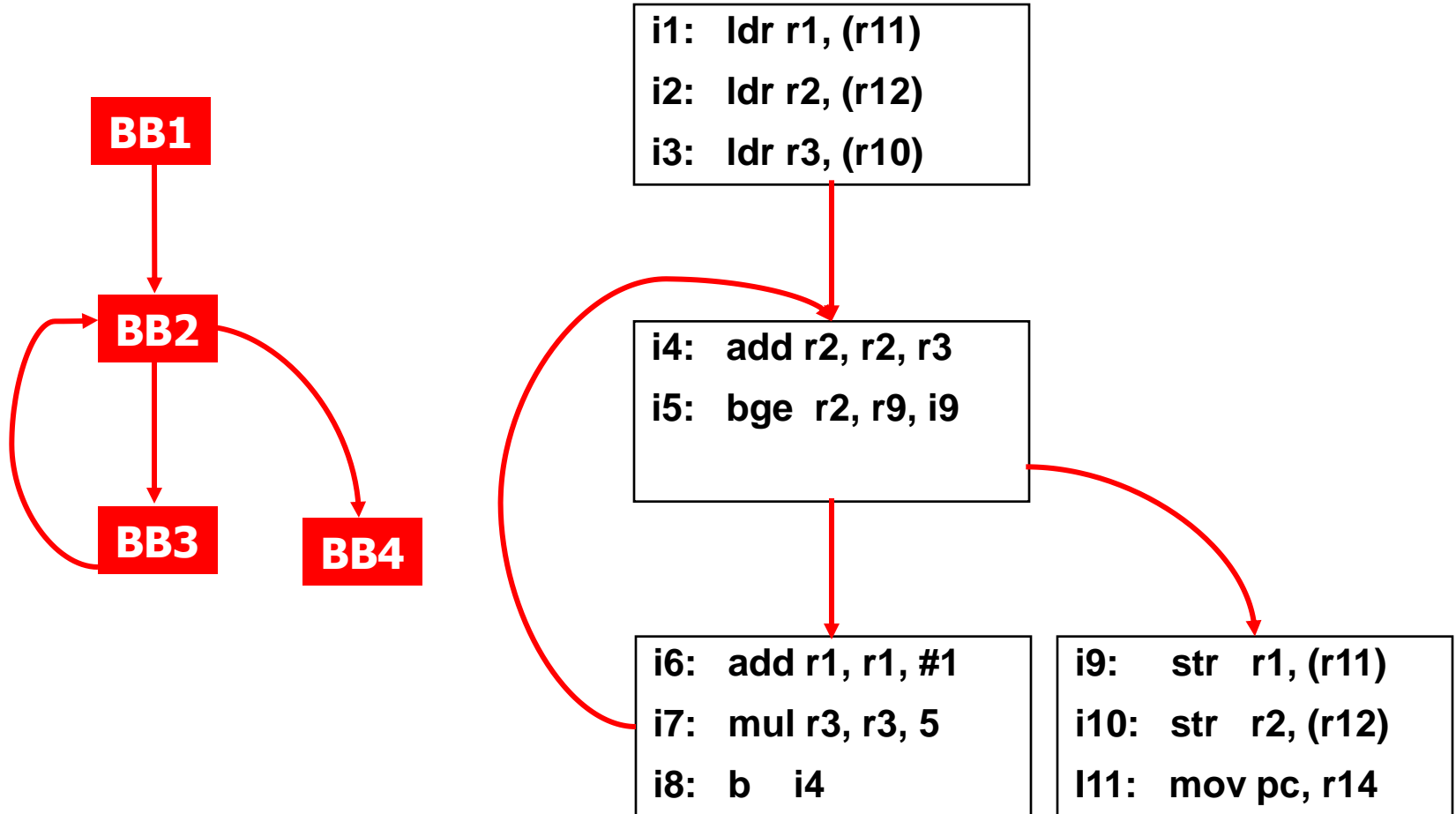
```
i1:  ldr  r1, (r11)  
i2:  ldr  r2, (r12)  
i3:  ldr  r3, (r10)  
i4:  add  r2, r2, r3  
i5:  bge  r2, r9, i9  
i6:  add  r1, r1, #1  
i7:  mul  r3, r3, 5  
i8:  b    i4  
i9:  str  r1, (r11)  
i10: str  r2, (r12)  
i11: mov  pc, r14
```

# Concept of Basic Blocks

```
a = array[i];  
b = array[j];  
c = array[k];  
d = b + c;  
while (d < t) {  
    a++;  
    c *= 5;  
    d = b + c;  
}  
array[i] = a;  
array[j] = d;
```



# Control Flow Graph



# ILP (without Speculation)

BB1

ldr r1, (r11)

ldr r2, (r12)

ldr r3, (r13)

BB1

i1: ldr r1, (r11)

i2: ldr r2, (r12)

i3: ldr r3, (r13)

BB2

add r2, r2, r3

bge r2, r9, i9

BB3

add r1, r1, #1

mul r3, r3, 5

b i4

BB4

str r1, (r11)

str r2, (r12)

mov pc, r14

BB2

i4: add r2, r2, r3

i5: bge r2, r9, i9

BB3

i6: add r1, r1, #1

i7: mul r3, r3, 5

i8: b i4

BB4

i9: str r1, (r11)

i10: str r2, (r12)

i11: mov pc, r14

BB1 → BB2 → BB3

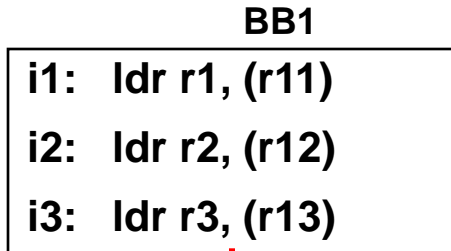
ILP = 8/4 = 2

BB1 → BB2 → BB4

ILP = 8/5 = 1.6

# ILP (with Speculation, No Control Dependence)

BB1 → BB2 → BB3

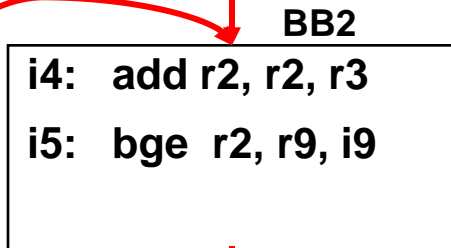


```
ldr r1, (r11)
add r2, r2, r3
bge r2, r9, i9
```

```
ldr r2, (r12)
add r1, r1, #1
b i4
```

```
ldr r3, (r13)
mul r3, r3, 5
```

ILP =  $8/3 = 2.67$



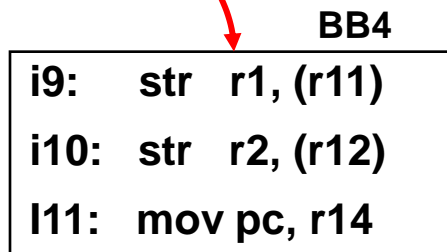
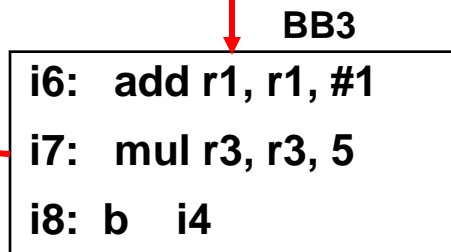
```
ldr r1, (r11)
add r2, r2, r3
bge r2, r9, i9
```

BB1 → BB2 → BB4

```
ldr r2, (r12)
str r1, (r11)
str r2, (r12)
```

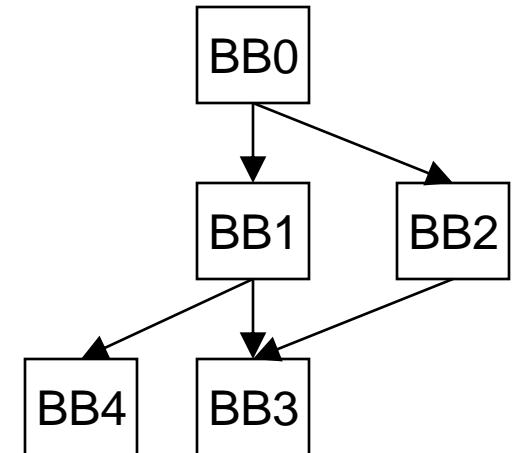
```
ldr r3, (r13)
mov pc, r14
```

ILP =  $8/3 = 2.67$



# Flynn's Bottleneck

- ILP  $\approx$  1.86 ☹
  - Programs on IBM 7090
  - ILP exploited **within basic blocks**
- [Riseman & Foster'72]
  - Breaking control dependency
  - A perfect machine model
  - Benchmark includes numerical programs, assembler and compiler



passed jumps	0 jump	1 jump	2 jumps	8 jumps	32 jumps	128 jumps	$\infty$ jumps
Average ILP	1.72	2.72	3.62	7.21	14.8	24.2	51.2

# [Wall'93]



- ⌘ Evaluating effects of microarchitecture on ILP
- ⌘ OOO with 2K instruction window, 64-wide, unit latency

models	branch predict	ind jump predict	reg renaming	alias analysis	ILP
Stupid	NO	NO	NO	NO	1.5 - 2
Poor	64b counter	NO	NO	peephole	2 - 3
Fair	2Kb ctr/gsh	16-addr ring no table	NO	Perfect	3 - 4
Good	16kb loc/gsh	16-addr ring 8-addr table	64 registers	perfect	5 - 8
Great	152 kb loc/gsh	2k-addr ring 2k-addr table	256	perfect	6 - 12
Superb	fanout 4, then 152kb loc/gsh	2k-addr ring 2k-addr table	256	perfect	8 - 15
Perfect	Perfect	Perfect	Perfect	perfect	18 - 50




# ILP in Embedded Systems

- Can save power by exploiting more ILP, eventually decreasing clock frequency

## ILP Roadblocks

- **Data dependence**
  - Read-after-Write (RAW) or flow dependency (true)
  - Write-after-Write (WAW) or output dependency (false)
  - Write-after-Read (WAR) or anti-dependency (false)
- **Control dependence**
  - Branch
    - Conditional
    - Indirect

# Exploiting ILP

- Hardware
  - Control speculation (control)
  - Dynamic Scheduling (data)
  - Register Renaming (data)
  - Dynamic memory disambiguation (data)
- Software  Many embedded system designers chose this
  - (Sophisticated) program analysis
  - Predication or conditional instruction (control)
  - Better register allocation (data)
  - Memory Disambiguation by compiler (data)

# A look at IPC

- IPC = instruction per cycle
- IPC  $\neq$  ILP
- IPC looks on the bottleneck on real machine with resource limited

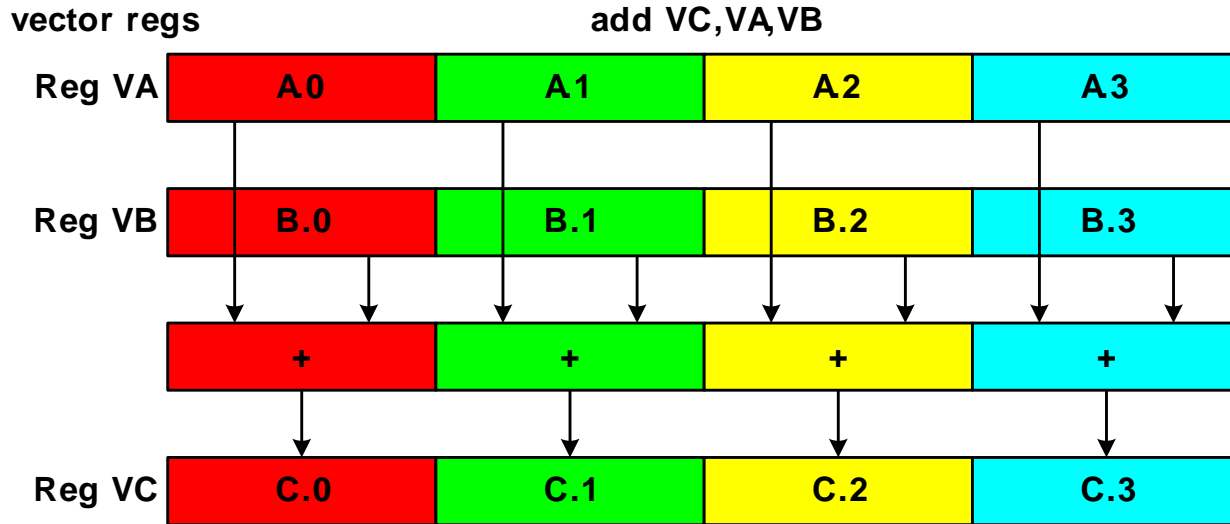
# Other Parallelisms

- SIMD (Single instruction, Multiple data)
  - Each register as a collection of smaller data
- Vector processing
  - e.g. VECTOR ADD: add long streams of data
  - Good for very regular code containing long vectors
  - Bad for irregular codes and short vectors
- Multithreading and Multiprocessing (or Multi-core)
  - Cycle interleaving
  - Block interleaving
  - High performance embedded's option (e.g., packet processing)
- Simultaneous Multithreading (SMT): *Hyper-threading*
  - Separate contexts, shared other microarchitecture modules

# SIMD Architecture

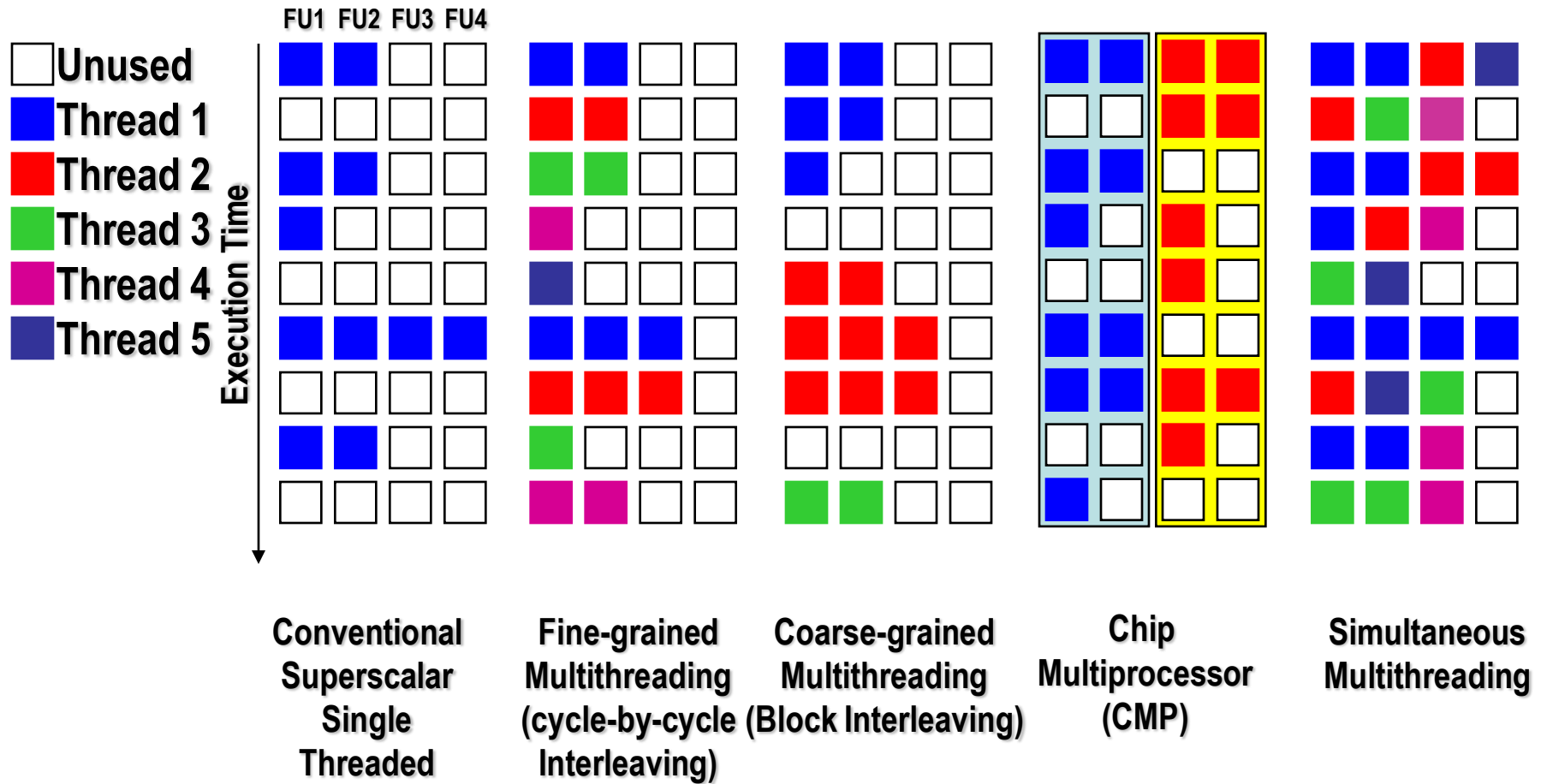
- SIMD = “single-instruction multiple-data”
- SIMD exploits data-level parallelism
  - a single instruction can apply the same operation to multiple data elements in parallel
- SIMD units employ “vector registers”
  - each register holds multiple data elements

# A SIMD Instruction Example



- Example is a 4-wide add
  - each of the 4 elements in reg VA is added to the corresponding element in reg VB
  - the 4 results are placed in the appropriate slots in reg VC

# Multithreading (MT) Paradigms



# Embedded processors in the market

- PIC
- AVR
- 8051
- 68HC11
- MAXQ
- 68000
- DSP
- ARM
- ATOM

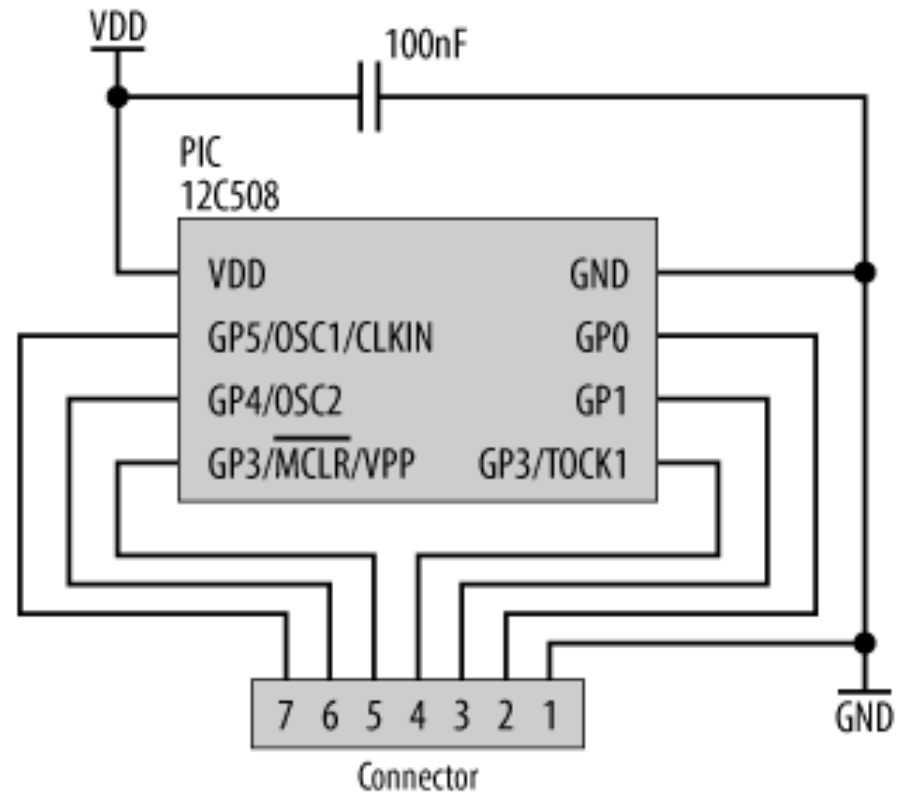


# PIC History

- In late 1970, General Instruments built 2 processor
  - 16-bit microprocessor named CP1600
  - I/O controller named Peripheral Interface Controller (PIC) for CP1600
- The company and its CP1600 died a quiet death
- PIC live-on! (in most game controllers and toys) under the company named Microchip
- Microchip is the number one in supplier of 8 bit-microcontroller

# Mini PIC12C805

- 8-bit RISC architecture
- 32 KHz
- 512 word memory
- Single cycle instructor



# Bigger PIC

- PIC16C73
- 4K of program memory
- Internal RAM
- SPI
- I<sup>2</sup>C
- UART
- 5 channels analog input



# AVR Microcontroller

- Developed in Norway by 2 students from Norwegian Institute of Technology
- Produced by Atmel in 1996
- 8-bit RISC Architecture
- Single cycle instruction
- Up to 20 MHz



# 8051 microcontroller

Developed by Intel

8-bit CISC architecture 16 bit addressing

Harvard architecture (separate instruction and data cache)

Support multiply, divider (take more time)

Run up to 150 MHz



# 68HC11

- Developed by Freescale Semiconductor (formerly Motorola)
- CISC architecture
- 8-bit microcontroller, 16-bit address indexing
- Widely used in Barcode-reader, automotive industry, and education
- Up to 50 MHz for 68HC12



# MAXQ

Developed by Dallas Semiconductor  
(subsidiary from Maxim ) in 2004

16-bit RISC architecture

Target for low power

Simple instruction and decoder(only mov  
instruction with accumulator)

16 bit processor, 32 MHz

# 68000

- 32 bit RISC microcontroller from Freescale
- Can address 16MB of memory
- 8 data register and 8 address register
- Can run 8 MHz – 20 MHz





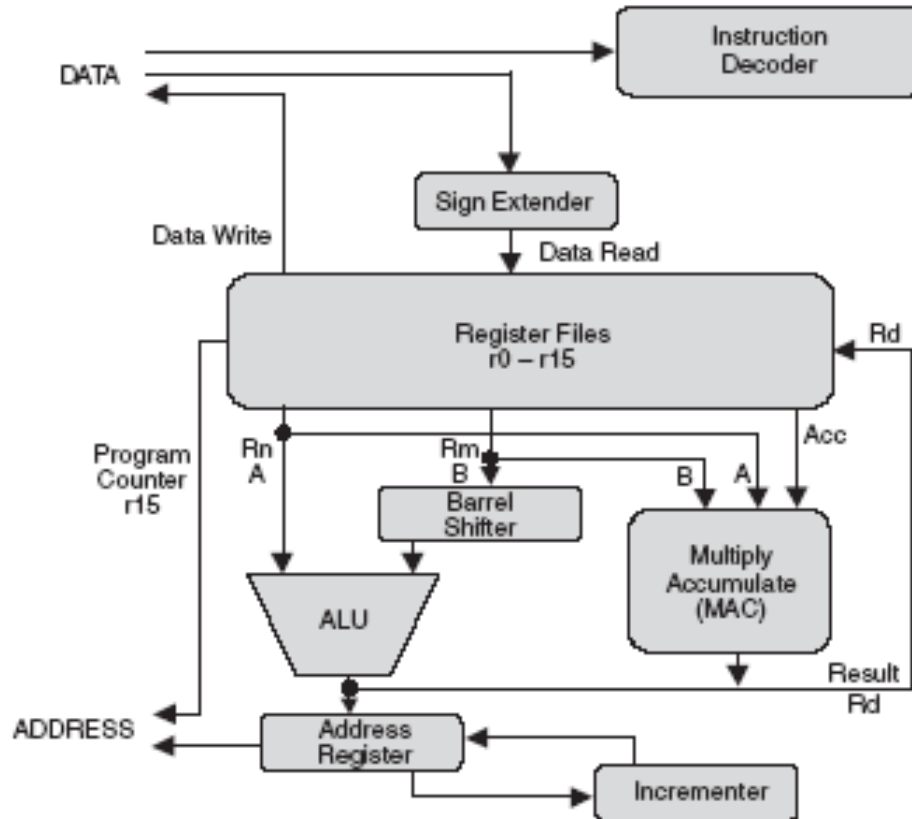
# DSP

- Major 3 companies
  - Texas Instrument (TMS320 series)
  - Analog devices (SHARC)
  - Freescale semiconductor (DSP56xx)
- Target signal processing applications
  - Radar
  - Speech processing
  - Video conference

# ARM 7 architecture

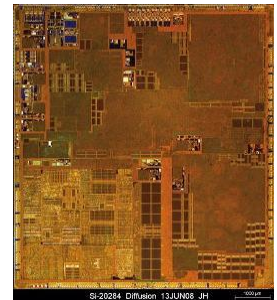
- 32-bit RISC architecture
- 3 stages pipeline:  
Fetch, Decode, Execute
- Target for low power

# ARM7 Architecture

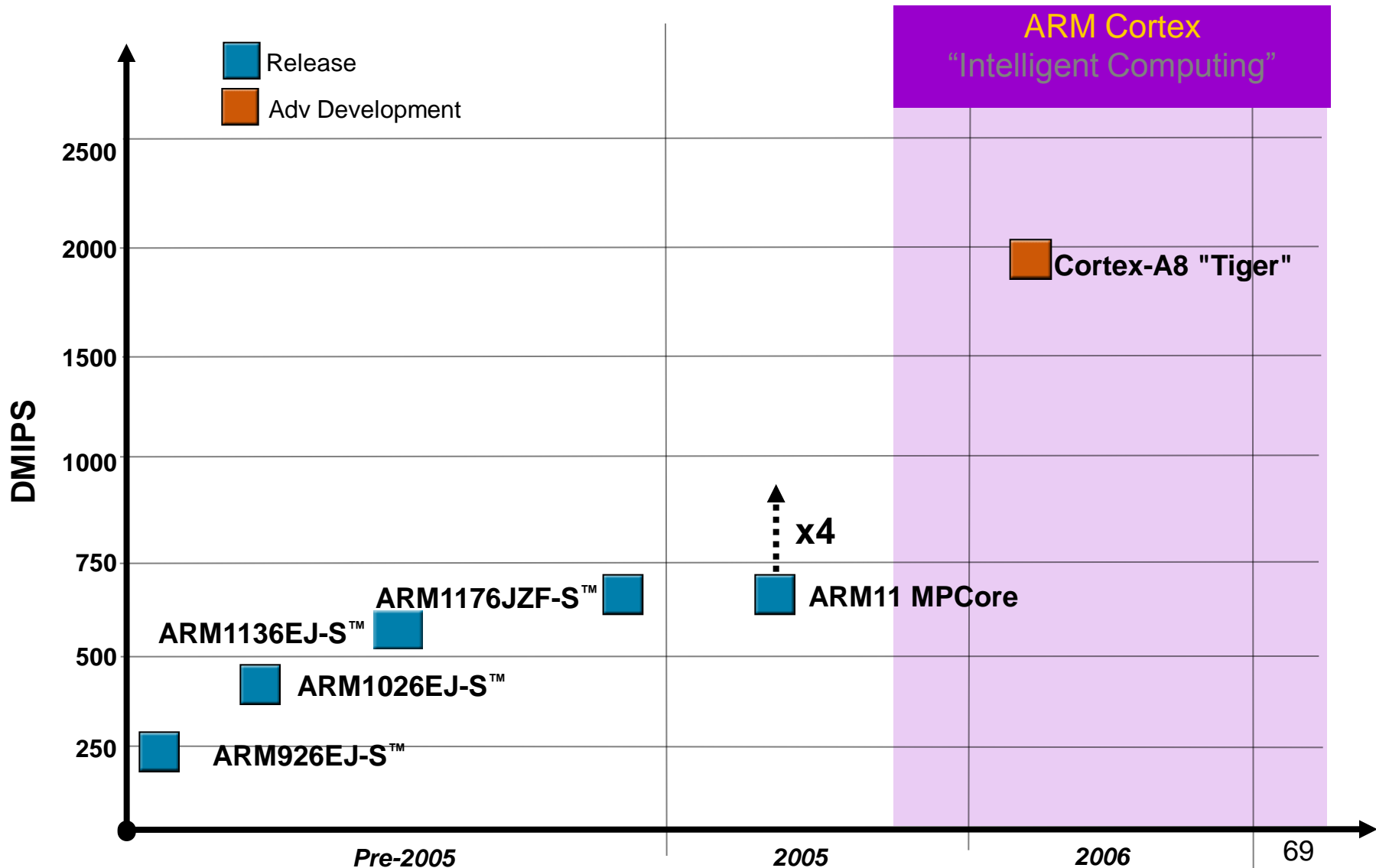


# ARM's Cortex-A8 (ARMv7-Cortex<sup>®</sup>)

- First implementation of ARMv7 ISA, including Advanced SIMD Media Extension (NEON) run at 600MHz – 1 GHz
- In-order, dual-issue superscalar core
  - 13-stage integer pipeline
  - 10-stage NEON media pipeline
  - Dedicated L2 with 9-cycle latency
  - Branch predictor based on global history
  - NEON: 64/128-bit SIMD, 2x-4x ↑ over prior ARMv6 SIMD
- Key metrics
  - Delivers 2000 Dhrystone MIPS for next-gen consumer apps
  - Average IPC of 0.9 across benchmark suites
    - EEMBC, SpecINT95, Mediabench, and vendor apps
  - Achieve 1 GHz when fab in high performance technology
  - Less than 300mW
  - Less than 4mm<sup>2</sup> at 65nm, excluding NEON, L2 cache, and Embedded Trace



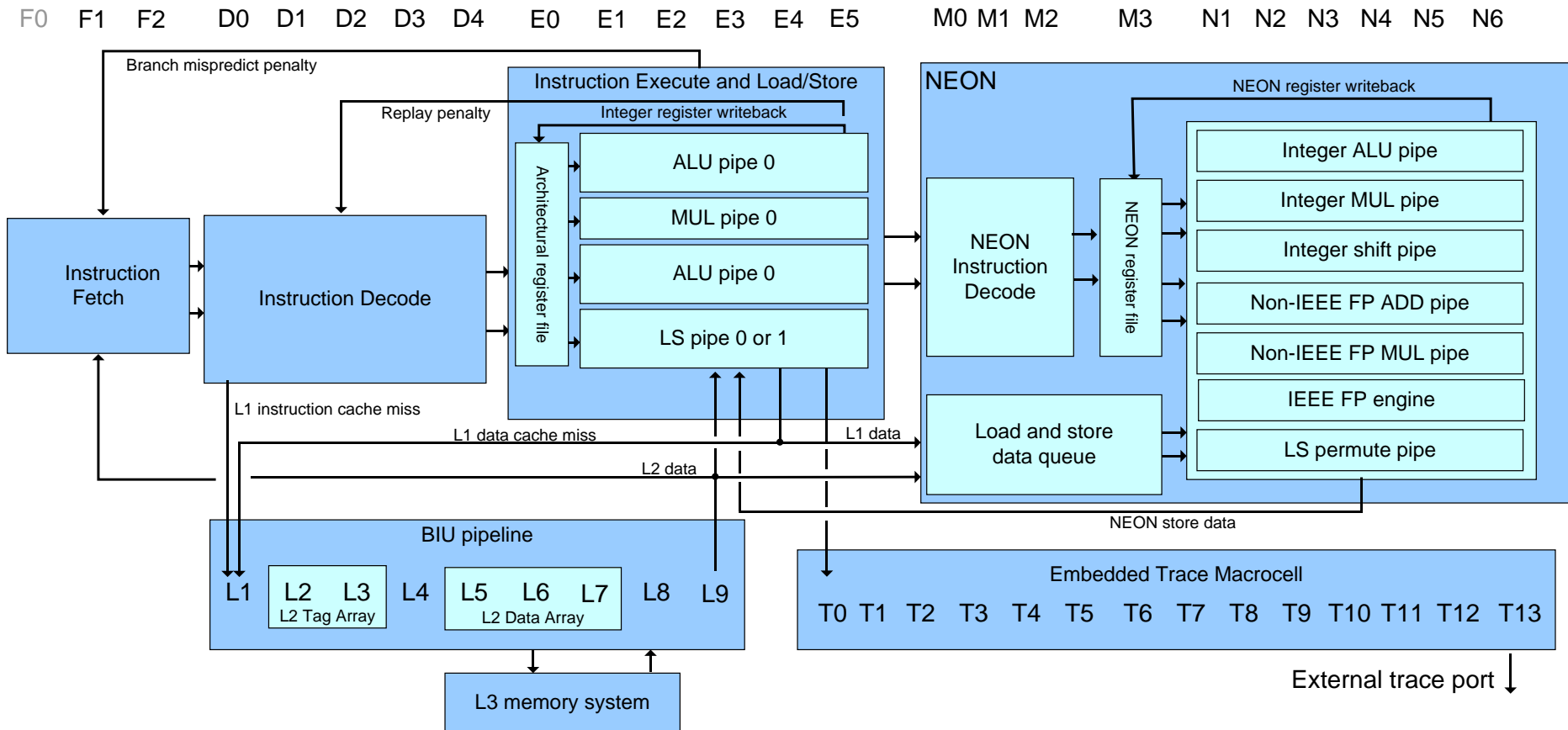
# ARM Core-Performance Roadmap



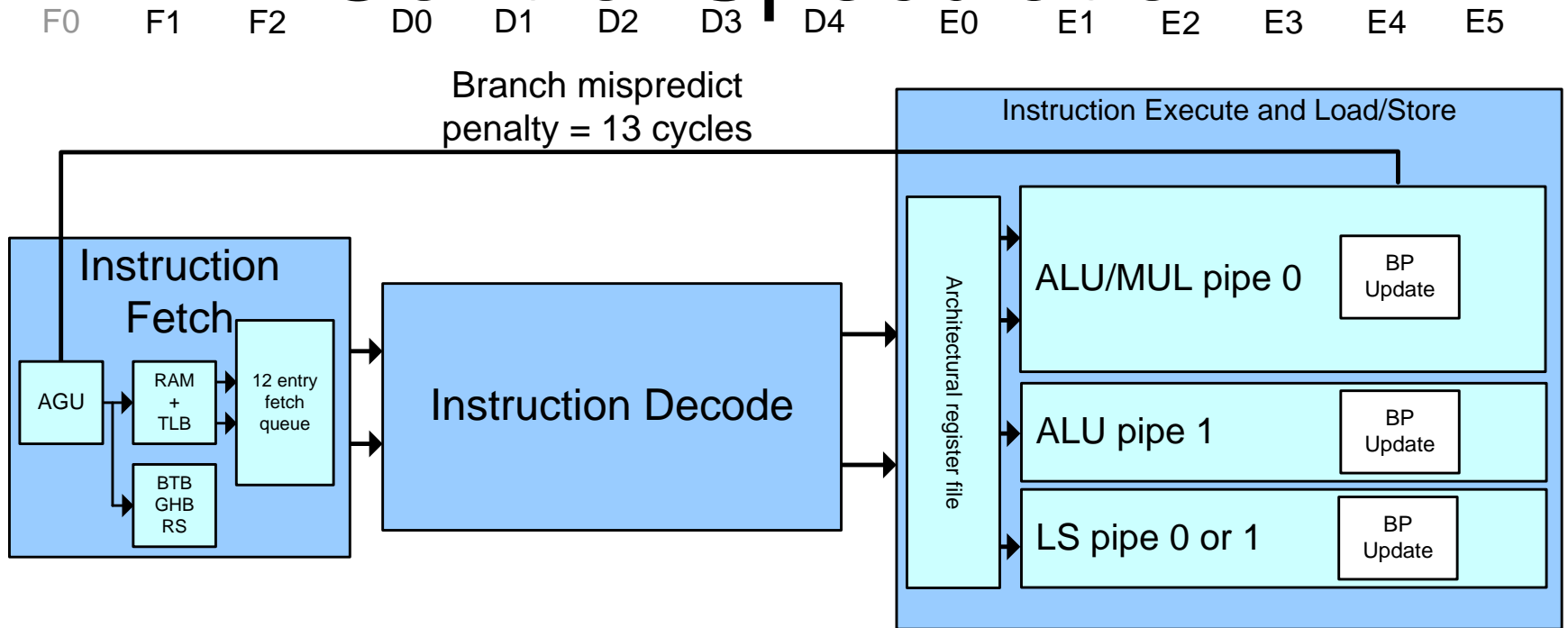
# Full Cortex-A8 Pipeline Design

## 13-Stage Integer Pipeline

## 10-Stage NEON Pipeline

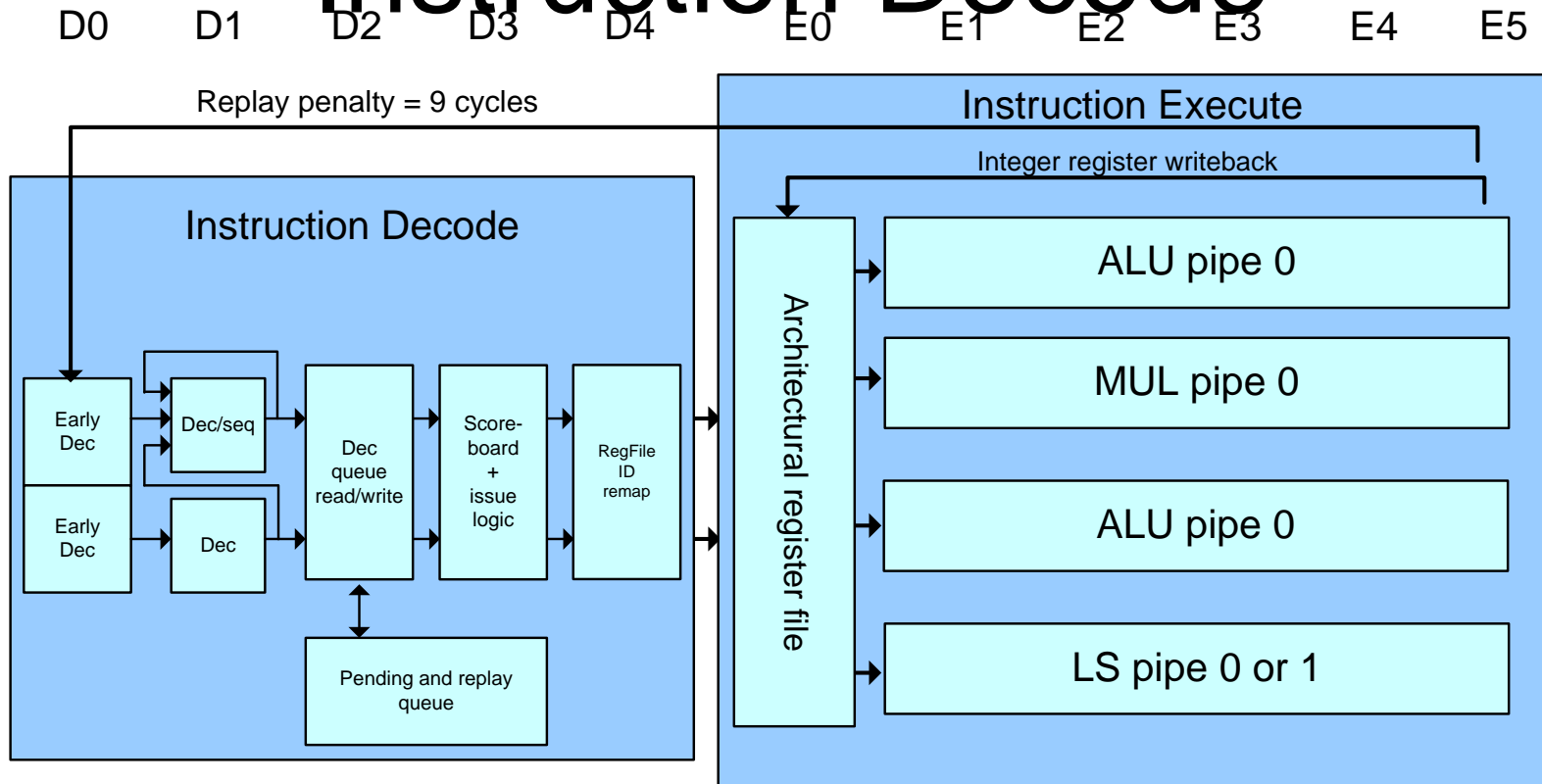


# Control Speculation



- Dynamic branch predictor
  - 512-entry 2-way BTB
  - 4K-entry GHB indexed by branch history and PC
  - 8-entry return stack
- Branch resolution
  - all branches are resolved in single stage
  - Maintains speculative and non-speculative versions of branch history and return stack

# Instruction Decode

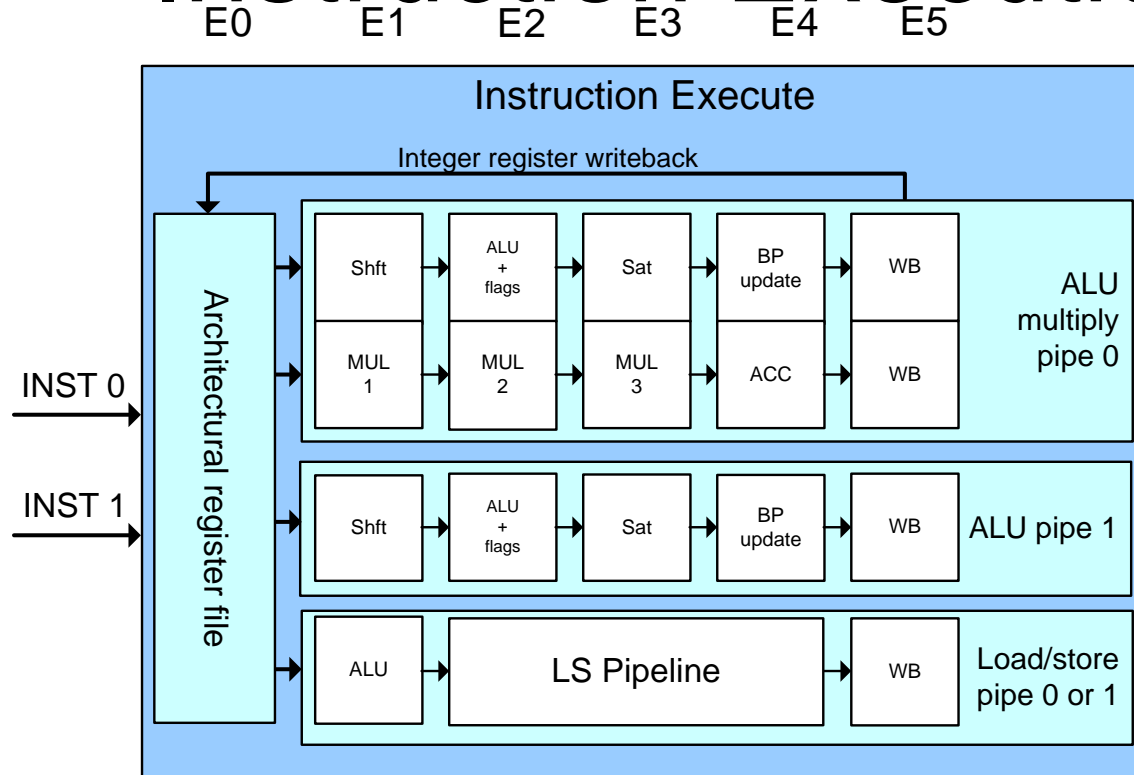


- **Instruction decode highlights**

- pending queue reduces Fetch stalls and increases pairing opportunities
- replay queue keeps instructions for reissue on memory system stall
- scoreboard predicts register availability using static scheduling techniques
- cross-checks in D3 allow issue of dependent instruction pairs

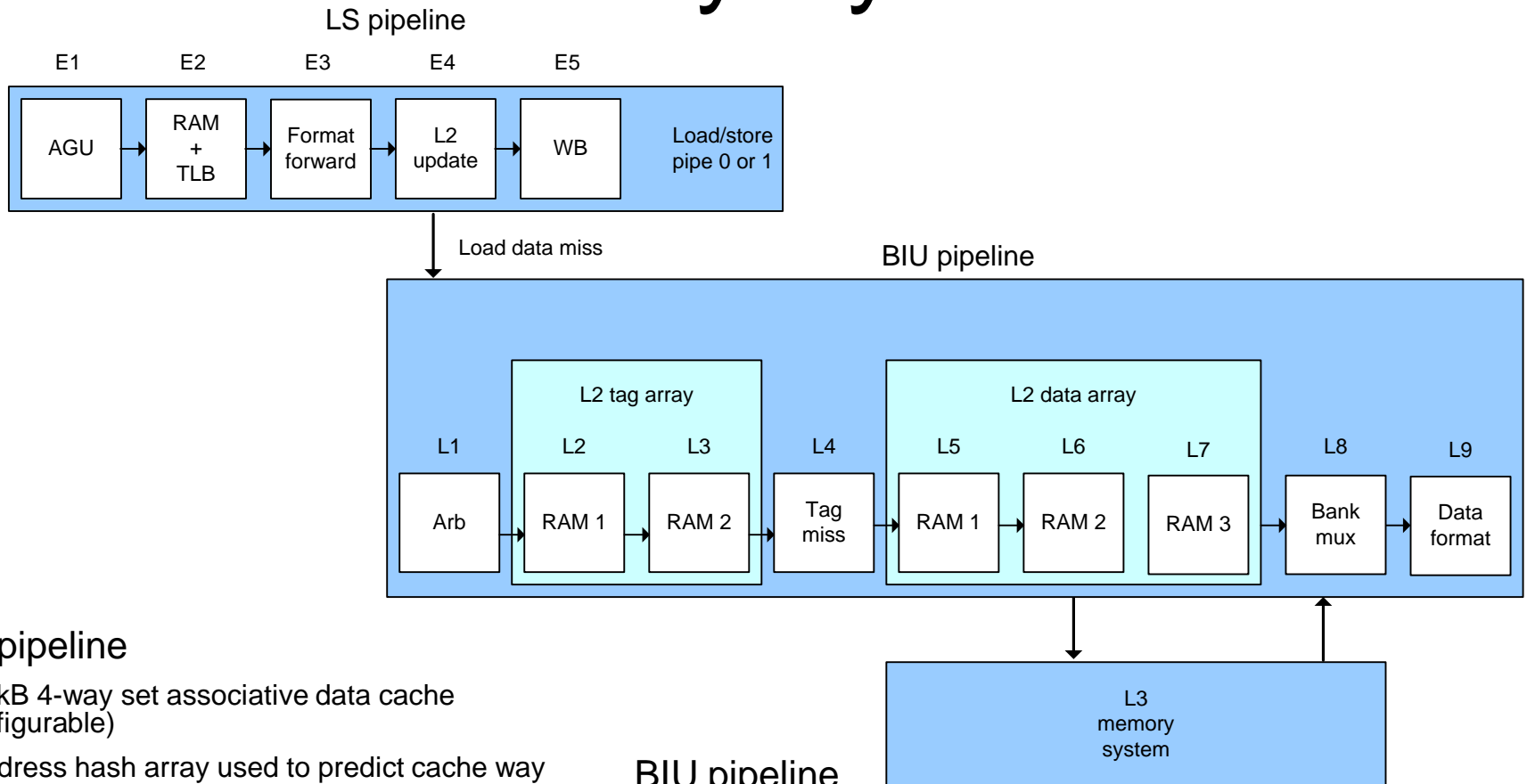


# Instruction Execution



- Execution pipeline highlights
  - 2 symmetric ALU pipelines: Shift/ALU/SAT
  - Load/store pipe used by instructions in either pipeline
  - Multiply instructions are tied to pipe 0
  - All key forwarding paths supported
  - Static scheduling allows for extensive clock gating

# Memory System



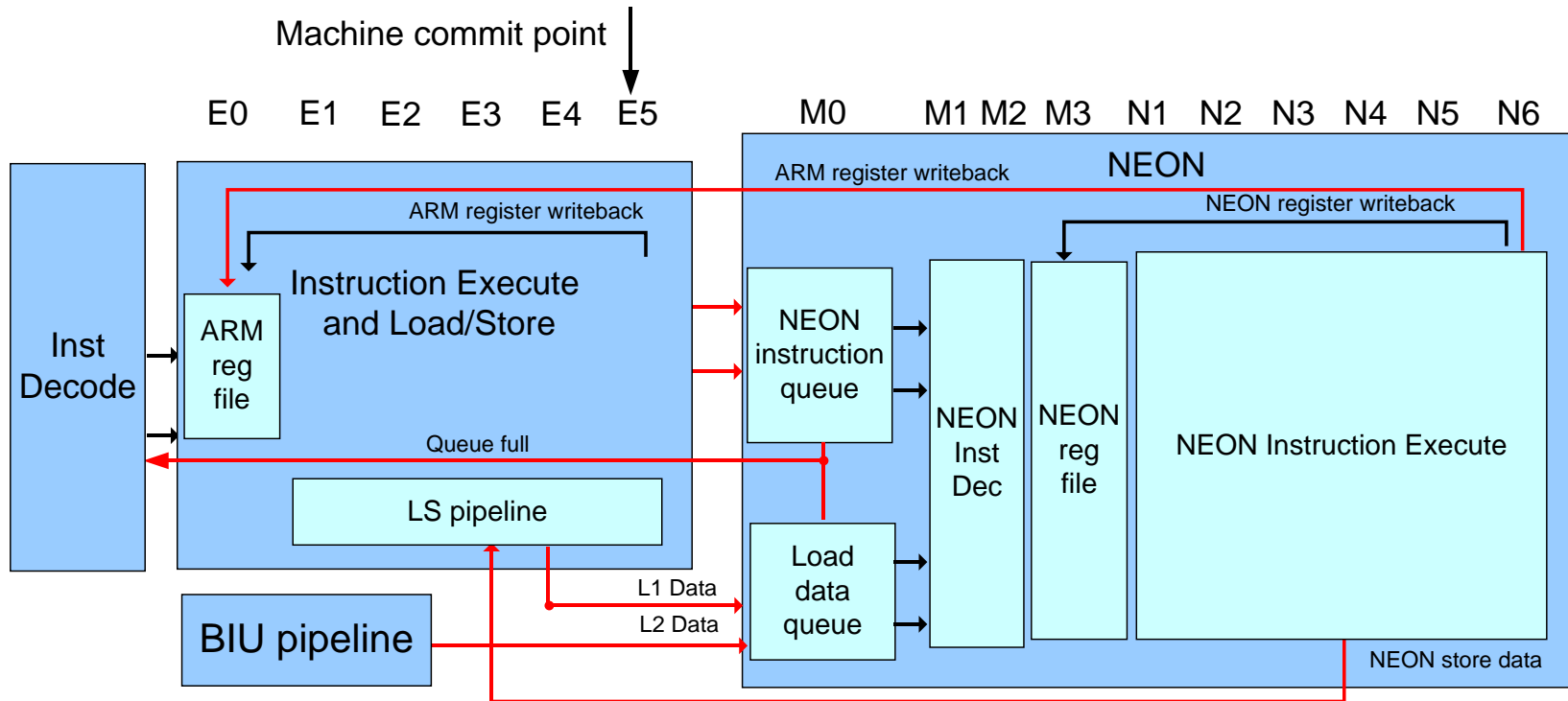
## LS pipeline

- 32kB 4-way set associative data cache (configurable)
- Address hash array used to predict cache way
  - Saves power and improves timing
- load data forwarding in E3 to all critical sources
  - one-cycle load-use penalty for ALU
- store data not required until E3

## BIU pipeline

- 9-cycle minimum access latency to L2 cache
- L2 built using standard compiled RAMS (64k-2MB configurable size)
- 64/128bit AXI L3 bus interface supports up to 9 outstanding transactions

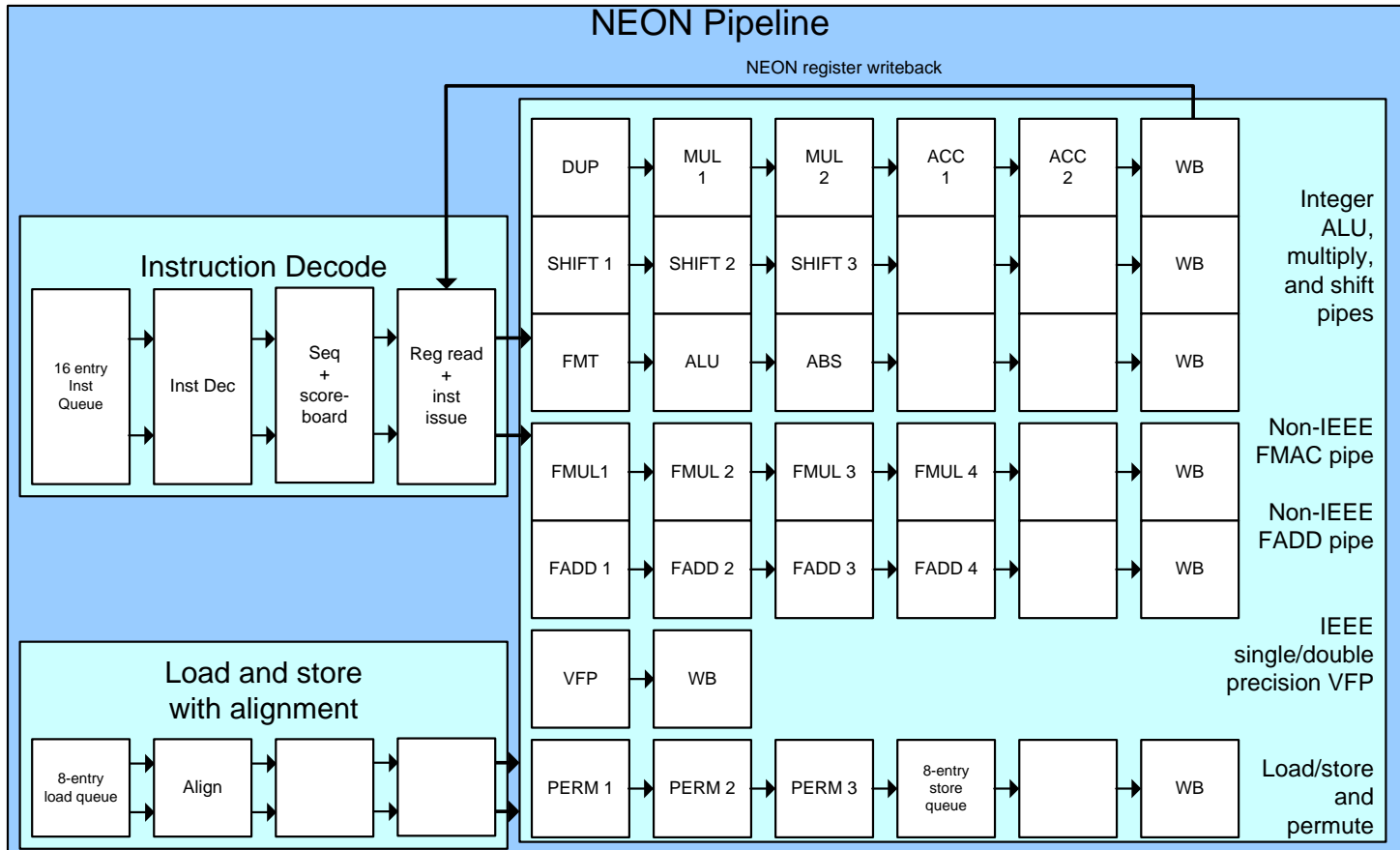
# NEON Interfaces



- Skewed late in pipeline, past the retire point
  - reduces interface complexity
    - exception handling not required
    - decoupling queues from integer machine
  - removes load-use penalty
  - negative impact on NEON -> ARM transfers
    - non-blocking ARM register file helps hide latency
- Streaming to and from L2 memory system
  - up to 8 outstanding transactions
  - can receive 128 bits/cycle
  - can receive data from L1 or L2 memory system
  - independent NEON store buffer

# NEON Media Engine Unit

M0 M1 M2 M3 N1 N2 N3 N4 N5 N6



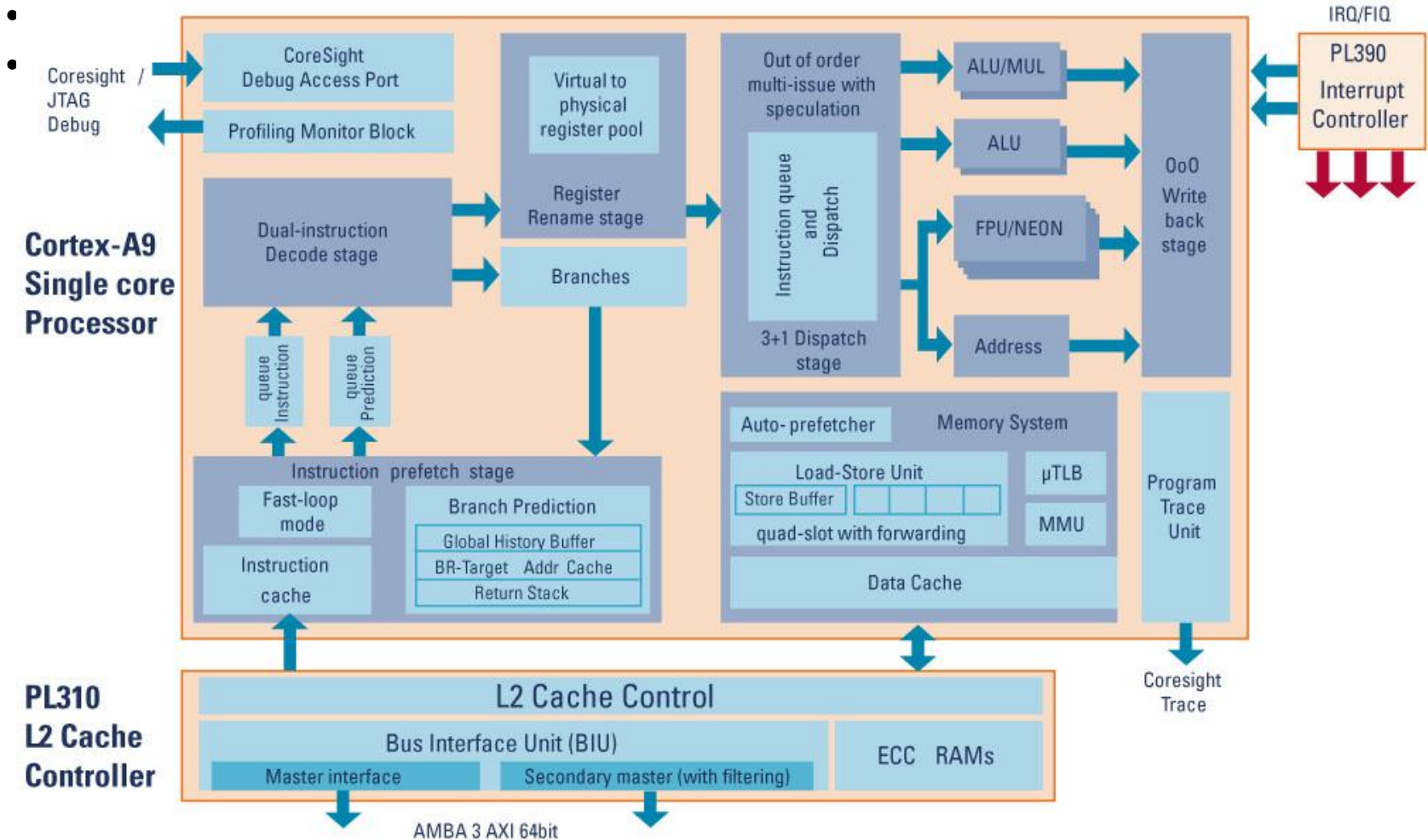
## Instruction issue

- static scheduling with fire-and-forget issue
- 1 LS + 1 NINT/NFP can issue each cycle

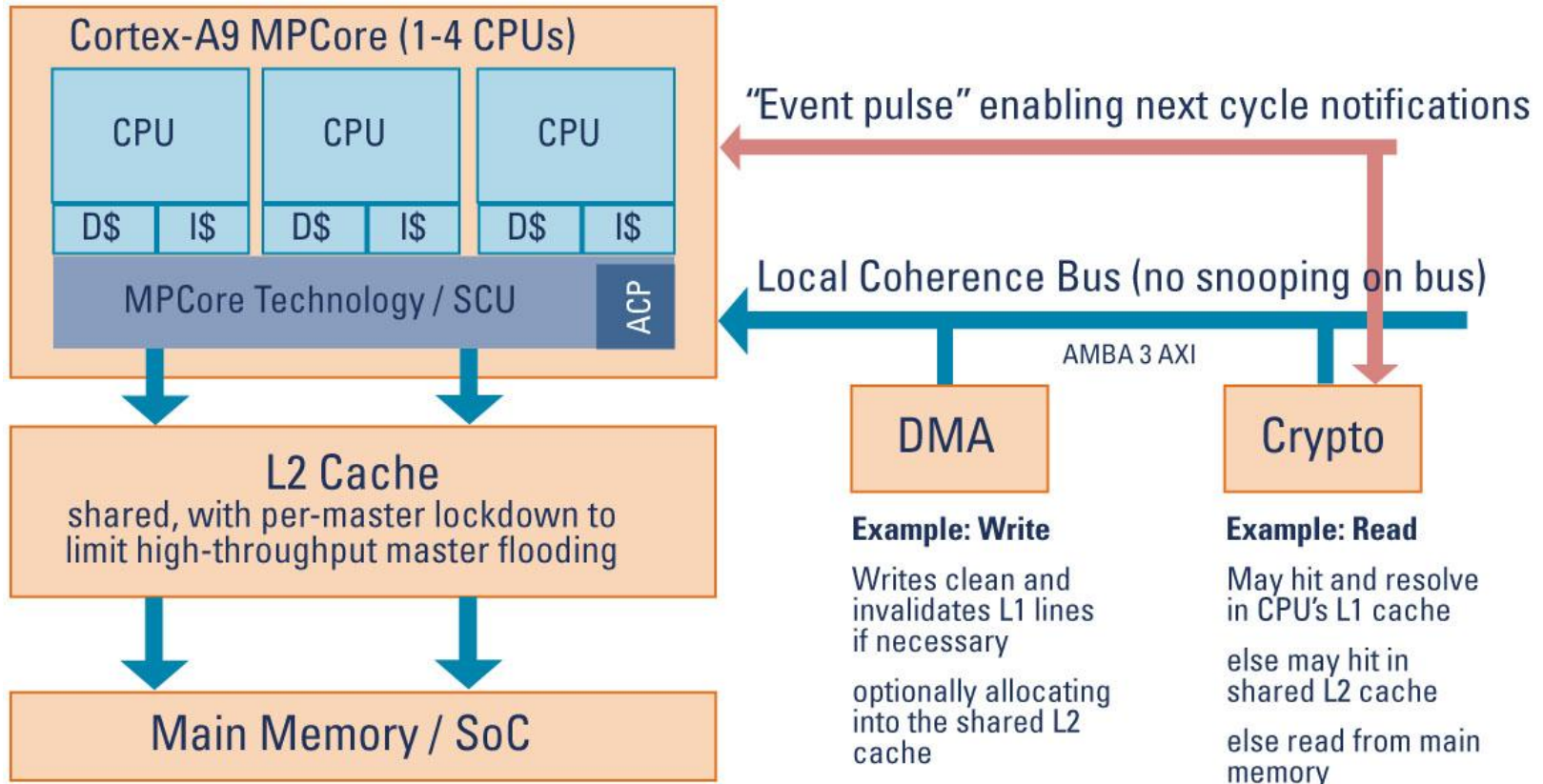
## Execution pipelines

- all pipelines are 64-bit SIMD
- floating-point MAC executed using both FADD and FMUL pipelines

# ARM's Cortex-A9

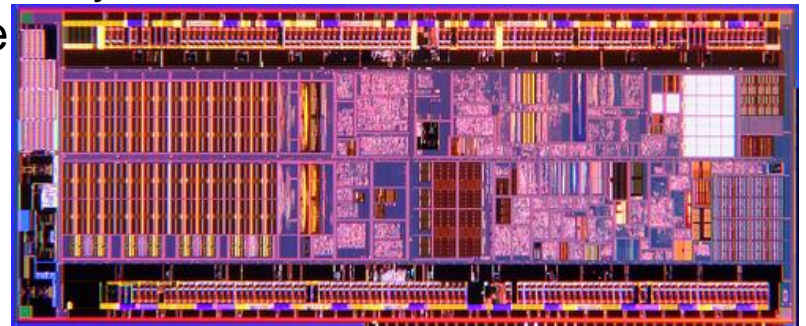


# Cortex-A9's Accelerator Coherence Port (ACP)



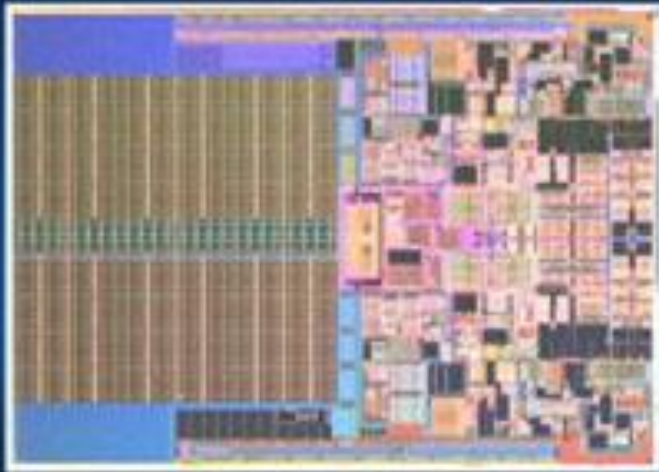
# Intel ATOM

- Code names: “Silverthorne” and “Diamondville”
- Design goals
  - High mobility
  - Low-power, low-power, low-power
  - Full x86 64-bit (Intel 64) compatibility
    - i.e., no software emulation, no broken legacy software
    - Support virtualization, SSSE3, SMT
    - Even run Windows Vista and definitely Linux
    - Ambitious or almost insane at the
  - Of course, cost
  - Almost from a clean-slate
- Markets
  - Netbooks (or sub-notebooks)
  - consumer electronics
  - MIDs



Silverthorne die, 45nm

# World's First Working 45 nm CPUs



## Penryn

45 nm Intel® Core™2  
family processor

*Mobile, desktop, workstation,  
and server applications*



## Silverthorne

45 nm Intel Ultra Low Power  
Processors

*For mobile internet devices  
and ultra mobile PCs*

Non-square shape,  
suggesting future  
dual-core





# ATOM Specifications

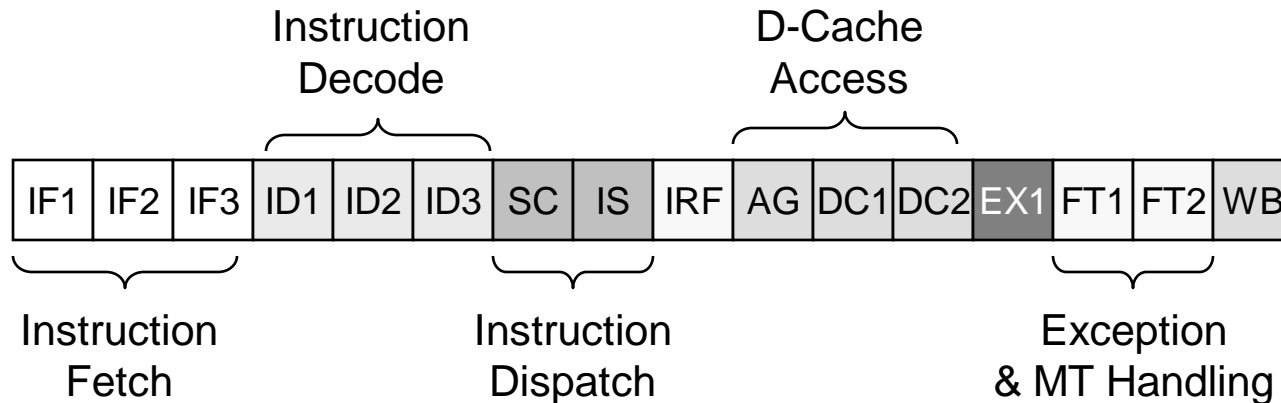
Feature	Intel Atom Z500	Intel Atom Z510	Intel Atom Z520	Intel Atom Z530	Intel Atom Z540
Core Freq	800MHz	1.10GHz	1.33GHz	1.60GHz	1.86GHz
FSB Freq	400MHz	400MHz	533MHz	533MHz	533MHz
Hyper-Threading	—	—	2 threads	2 threads	2 threads
L2 Cache	512K	512K	512K	512K	512K
TDP	650mW	2.0W	2.0W	2.0W	2.4W
Avg Power	160mW	220mW	220mW	220mW	220mW
Idle Power (C6)	80mW	100mW	100mW	100mW	100mW
Die Size	7.8mm x 3.1mm	7.8mm x 3.1mm	7.8mm x 3.1mm	7.8mm x 3.1mm	7.8mm x 3.1mm
Package	mFCBGA	mFCBGA	mFCBGA	mFCBGA	mFCBGA
Price*	\$45	\$45	\$65	\$95	\$160

- Operating voltage: 0.75V to 1.2V
- Test frequency up to 2 GHz
- TDP is the worst case, Avg power and idle power are more typical
- Points of comparison
  - Intel Celeron's TDP was 8W to 12W for their ULV part
  - VIA's Isaiah cannot match ATOM's TDP

# ATOM Fact

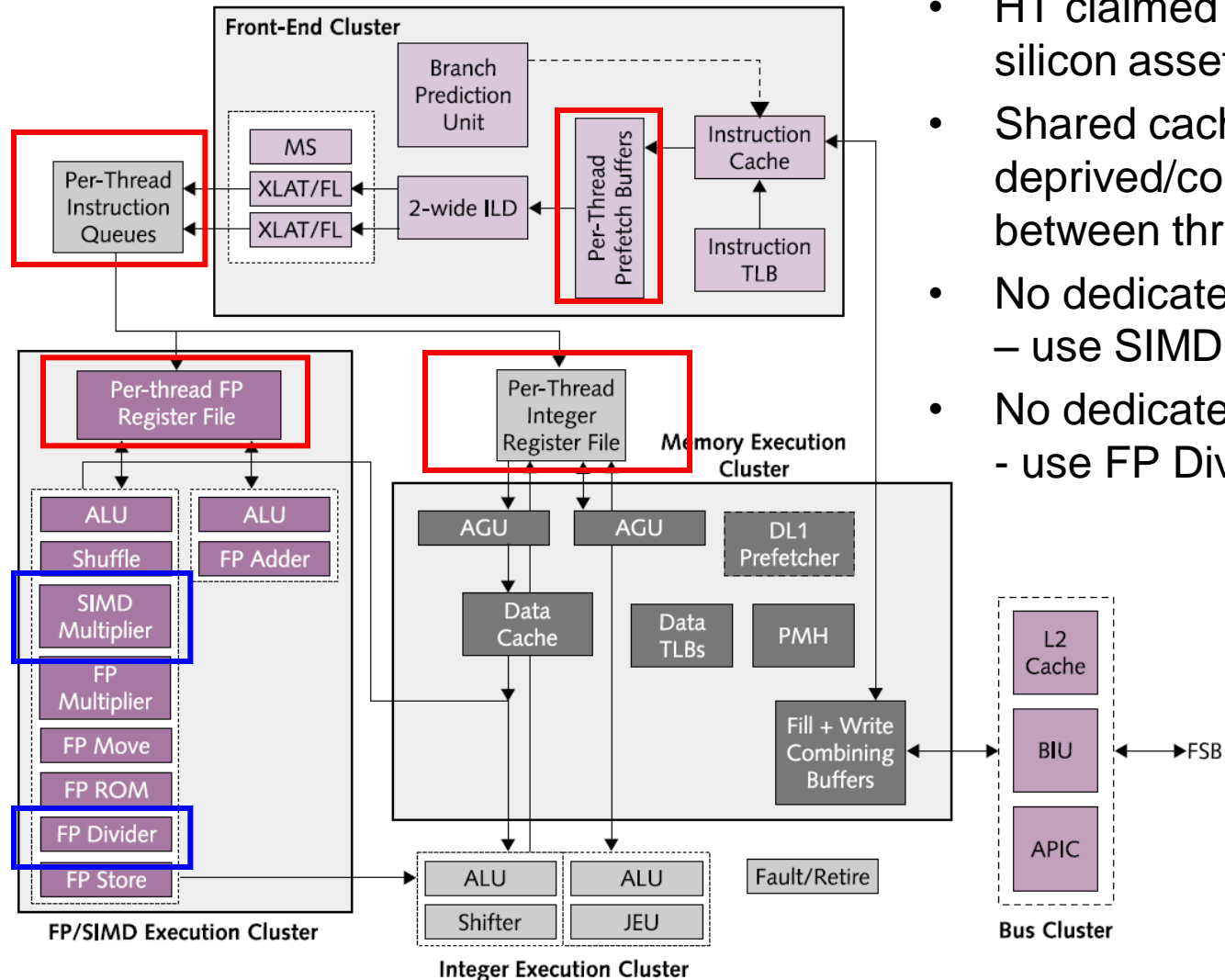
- Dimension: 3.1 x 7.8 mm (24.2 mm<sup>2</sup>)
  - VIA's Isaiah: 63 mm<sup>2</sup>
  - VIA's Centaur C7-M: 30 mm<sup>2</sup>, previous x86 record holder
- 47.1 million transistors @ 45nm, 9 metal layers
  - CPU core has 13.8 million transistors
- 16 pipeline stages
- Two-way superscalar in-order with Hyper-Threading
- L1 caches 24 kb data cache, 32 kb instruction cache
  - 8T cell with one read port and one write port
  - Operate at lower voltage
- L2 cache 512 kb cache
  - 6T SRAM cell
  - SECDED ECC protection implemented
  - Set associativity is programmable from 2 to 8 ways
- FSB
  - At 400MHz or 533 MHz
  - can be shut down

# ATOM Instruction Pipeline



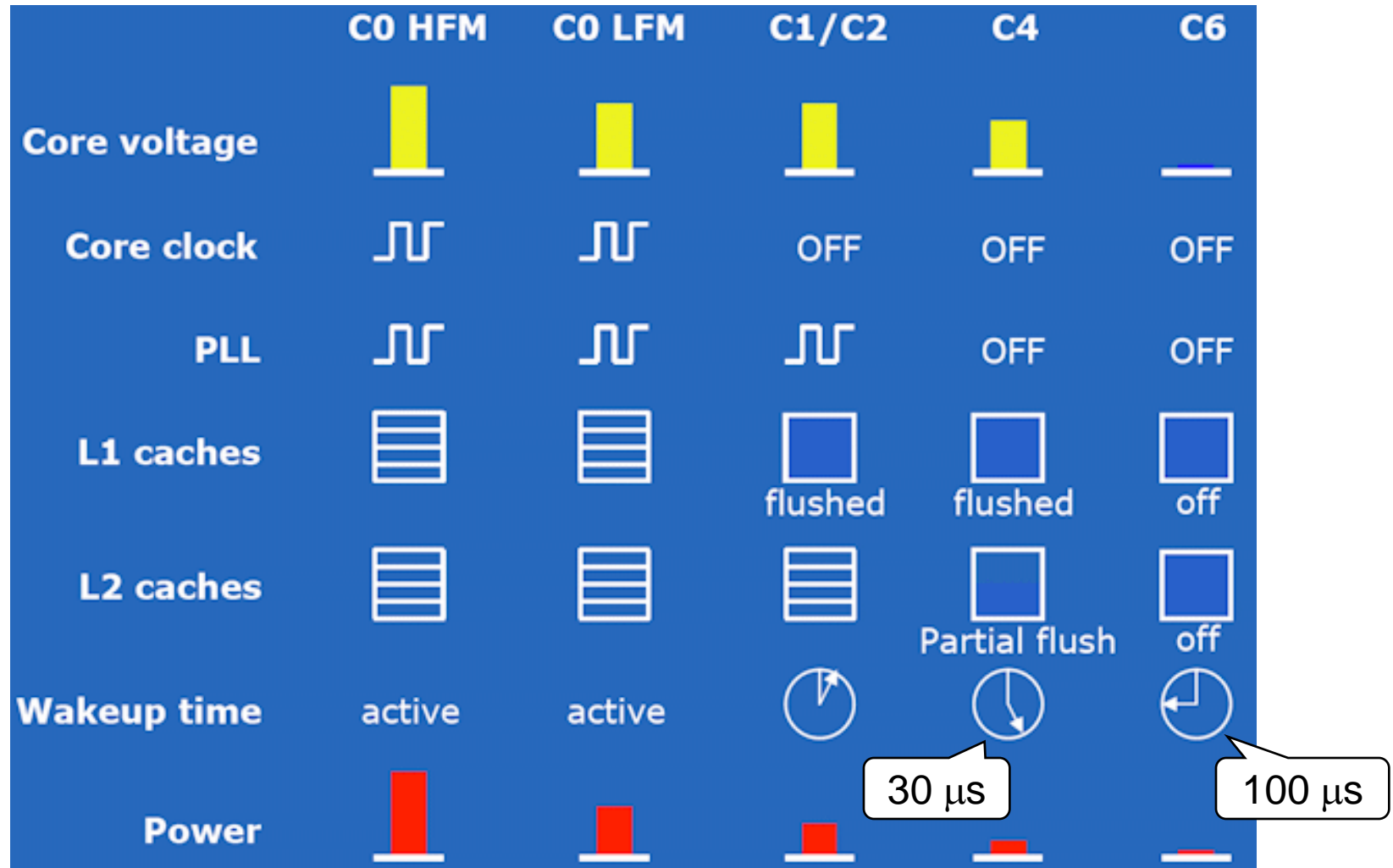
- To conserve “Power,” ATOM processor
  - Tagging boundaries in I-cache (i.e., skip variable length decoder)
    - X86 instruction can be 1 to 15 bytes
  - Decode assumes hit in the cache (otherwise, 19 stages)
  - No micro-op breakup (micro-op fusion), back to P54C
  - Discard aggressive control speculation
- Branch misprediction penalty 13 cycles

# ATOM Microarchitecture



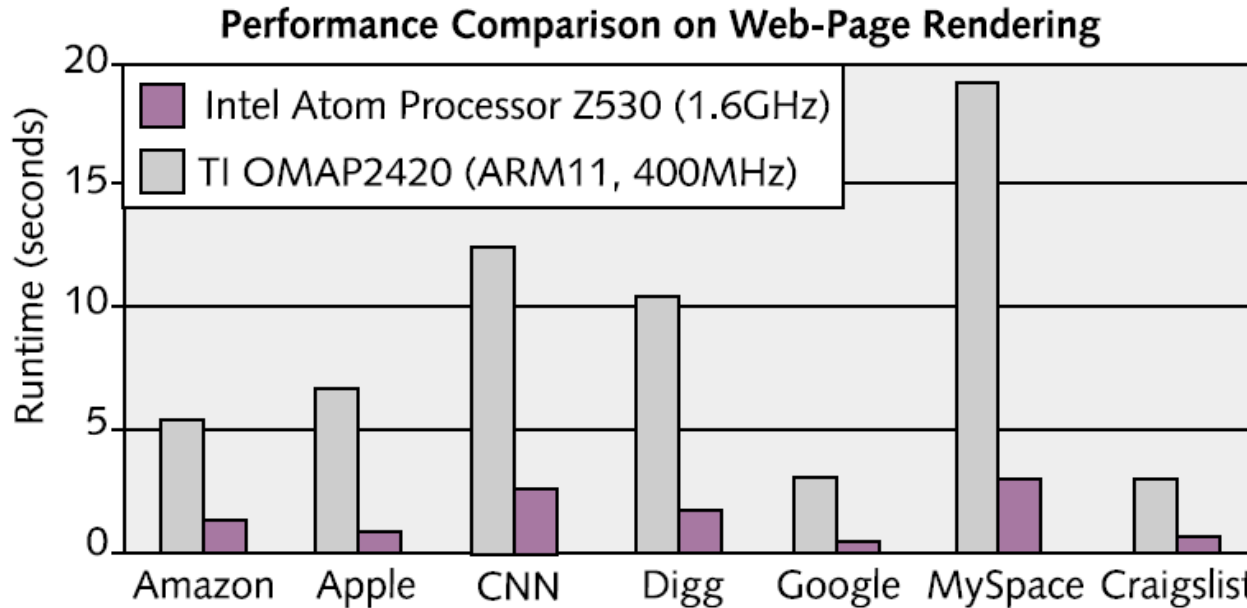
- HT claimed to enlarge silicon asset by 8%
- Shared cache space deprived/competed between threads
- No dedicated Multiplier – use SIMD Multiplier
- No dedicated Int Divider - use FP Divider

# Enhanced Speed States (C Steps)



- 0.875V drawing less than 1W running Vista

# Performance Comparison



Source: Microprocessor Report

- Webpages are on local flash (i.e., no network latency)
- ARM11
  - fully synthesized @ 90nm
  - Lower power (0.6mW/MHz → 400MHz ~ 240mW)
- ATOM's HT Technology
  - Improve performance by 36 to 47%
  - Worsen power by 17 to 19%

# Question?

You are chosen a processor for an airbag system. The program has 1000 instructions. Assumed that all processor has same ISA and IPC of 1. Each instruction takes 1 byte. The program has to finish execution within 1ms. Which processor from below table should be used, assuming that all architecture is byte-accessible?

- a) PIC 8 bit processor, 32 KHz, 10 baht
- b) AVR 8 bit processor, 20 MHz, 50 baht
- c) 68HC12 16 bit processor, 20 MHz, 100 baht
- d) ARM 32 bit processor, 500 MHz, 1000 baht