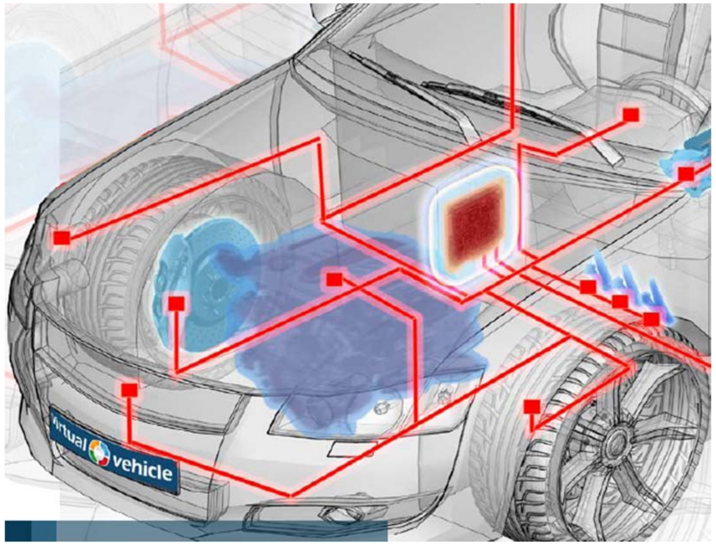
Embedded Systems

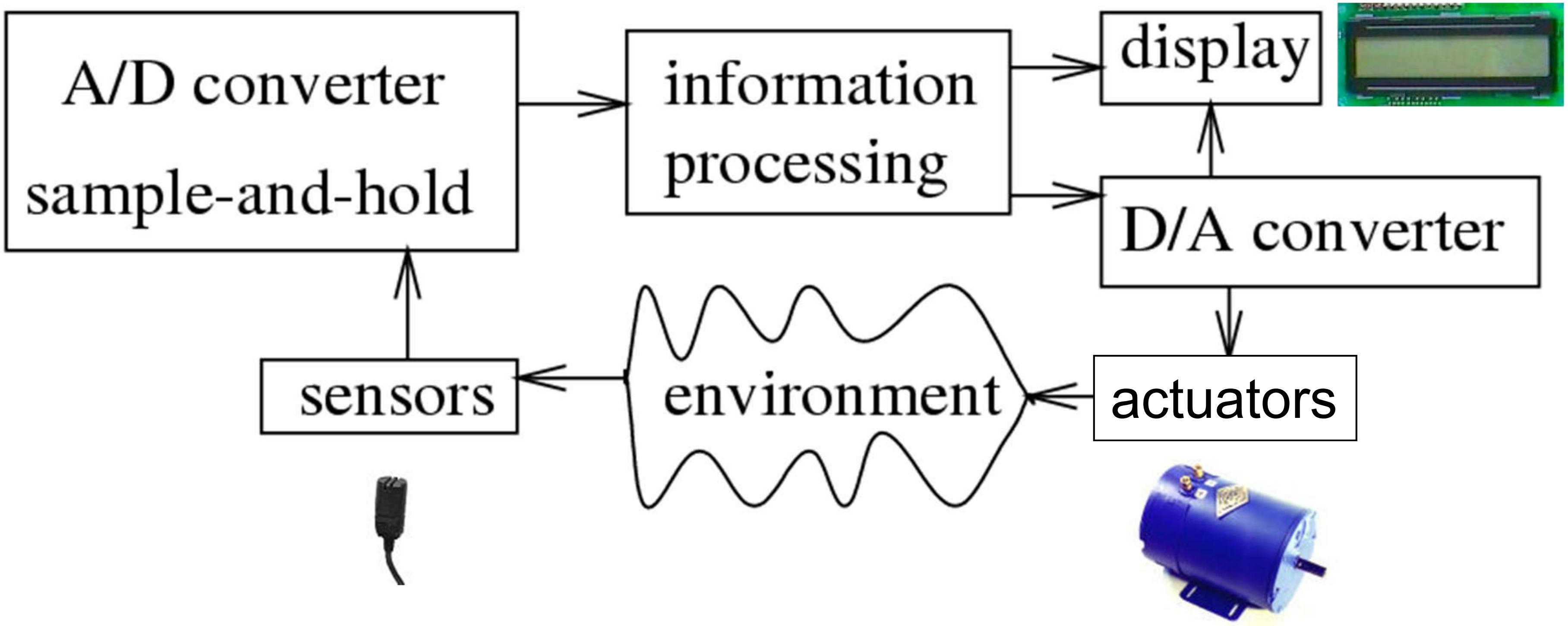


Dr. Eric Armengaud from the Virtual Vehicle Competence Center is going to give a talk on **model-based development and test of distributed automotive embedded systems** on **Tuesday, Jan. 11th**.

- Automotive embedded Systems
- SW Engineering
- networks (focus FlexRay)

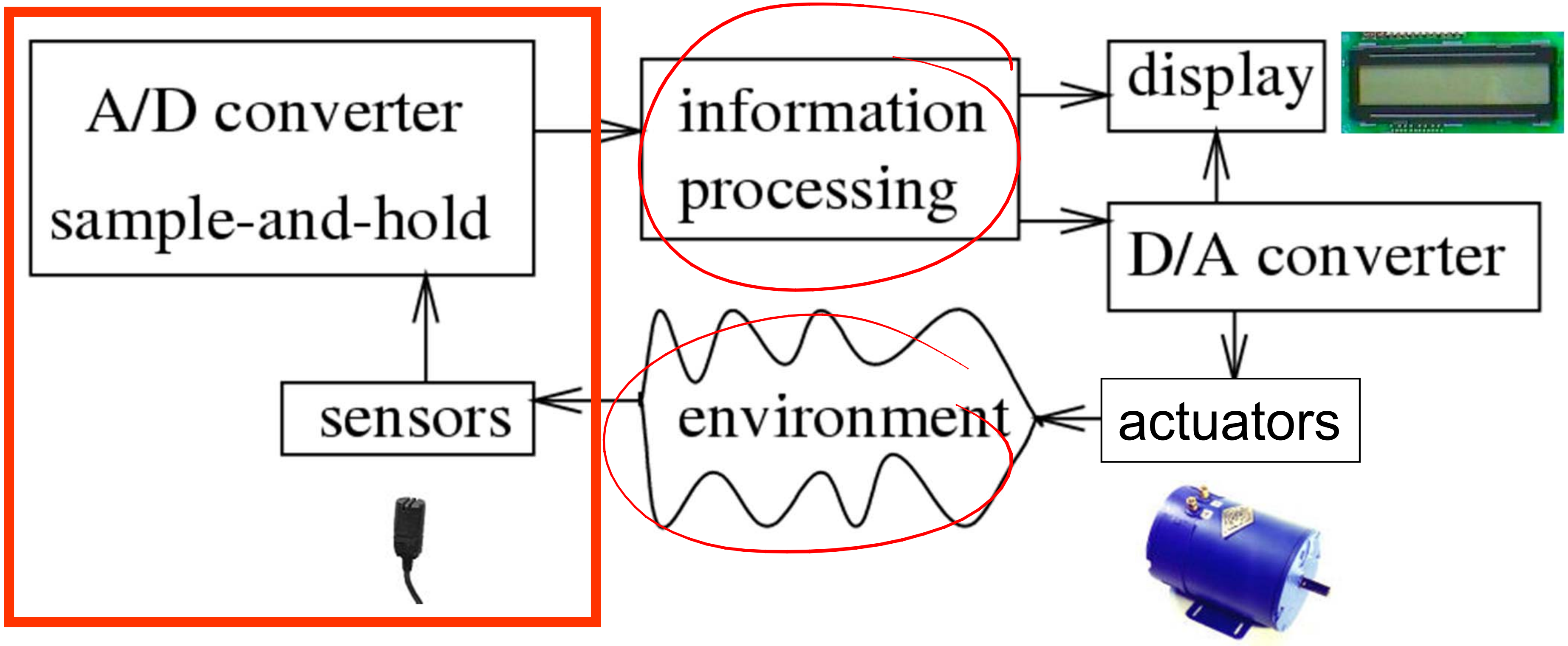
Embedded System Hardware

- Embedded system hardware is frequently used in a loop („***hardware in a loop***“):



Embedded System Hardware

- Embedded system hardware is frequently used in a loop („***hardware in a loop***“):



TI Embedded Processing Portfolio

TI Embedded Processors

Microcontrollers (MCUs)			ARM®-Based Processors	Digital Signal Processors (DSPs)		
16-bit ultra-low power MCUs	32-bit real-time MCUs	32-bit ARM Cortex™-M3 MCUs	ARM Cortex-A8 MPUs	DSP DSP+ARM	Multi-core DSP	Ultra Low power DSP
MSP430™	**C2000™ Delfino™ Piccolo™**	**Stellaris®** ARM® Cortex™-M3	**Sitara™** ARM® Cortex™-A8 & ARM9	**C6000™ DaVinci™** video processors **OMAP™**	**C6000™**	**C5000™**
Up to 25 MHz	40MHz to 300 MHz	Up to 100 MHz	300MHz to >1GHz	300MHz to >1Ghz +Accelerator	24.000 MMACS	Up to 300 MHz +Accelerator
Flash 1 KB to 256 KB	Flash, RAM 16 KB to 512 KB	Flash 8 KB to 256 KB	Cache, RAM, ROM	Cache RAM, ROM	Cache RAM, ROM	Up to 320KB RAM Up to 128KB ROM
Analog I/O, ADC LCD, USB, RF	PWM, ADC, CAN, SPI, I²C	USB, ENET MAC+PHY CAN, ADC, PWM, SPI	USB, CAN, PCIe, EMAC	USB, ENET, PCIe, SATA, SPI	SRIO, EMAC DMA, PCIe	USB, ADC McBSP, SPI, I²C
Measurement, Sensing, General Purpose	Motor Control, Digital Power, Lighting, Ren. Enrgy	Connectivity, Security, Motion Control, HMI, Industrial Automation	Industrial computing, POS & portable data terminals	Floating/Fixed Point Video, Audio, Voice, Security, Confer.	Telecom T&M, media gateways, base stations	Audio, Voice Medical, Biometrics
$0.25 to $9.00	$1.50 to $20.00	$1.00 to $8.00	$5.00 to $20.00	$5.00 to $200.00	$40 to $200.00	$3.00 to $10.00

Software & Dev. Tools

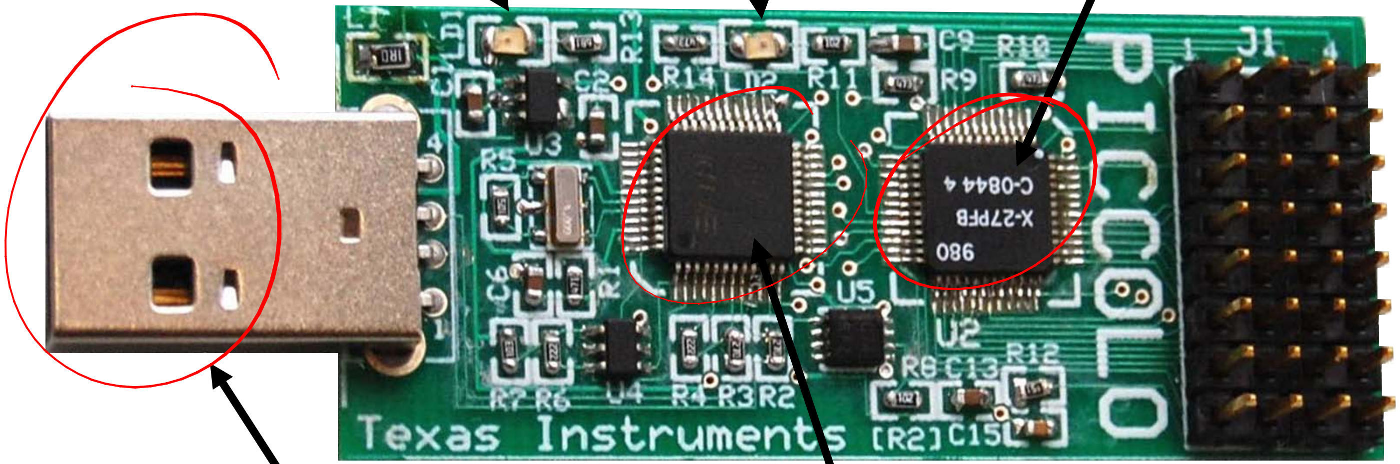
Texas Instruments Developer Network — Texas Instruments eXpressDSP™

Piccolo™ controlSTICK

LED LD1
(Power)

LED LD2
(GPIO34)

TMS320F28027
48-Pin Package



USB JTAG
Interface & Power

On-board USB
JTAG Emulation

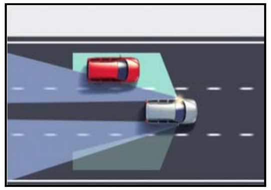
Peripheral
Header Pins

Broad C2000 Application Base

Renewable Energy Generation

Telecom Digital Power

AC Drives, Industrial & Consumer Motor Control



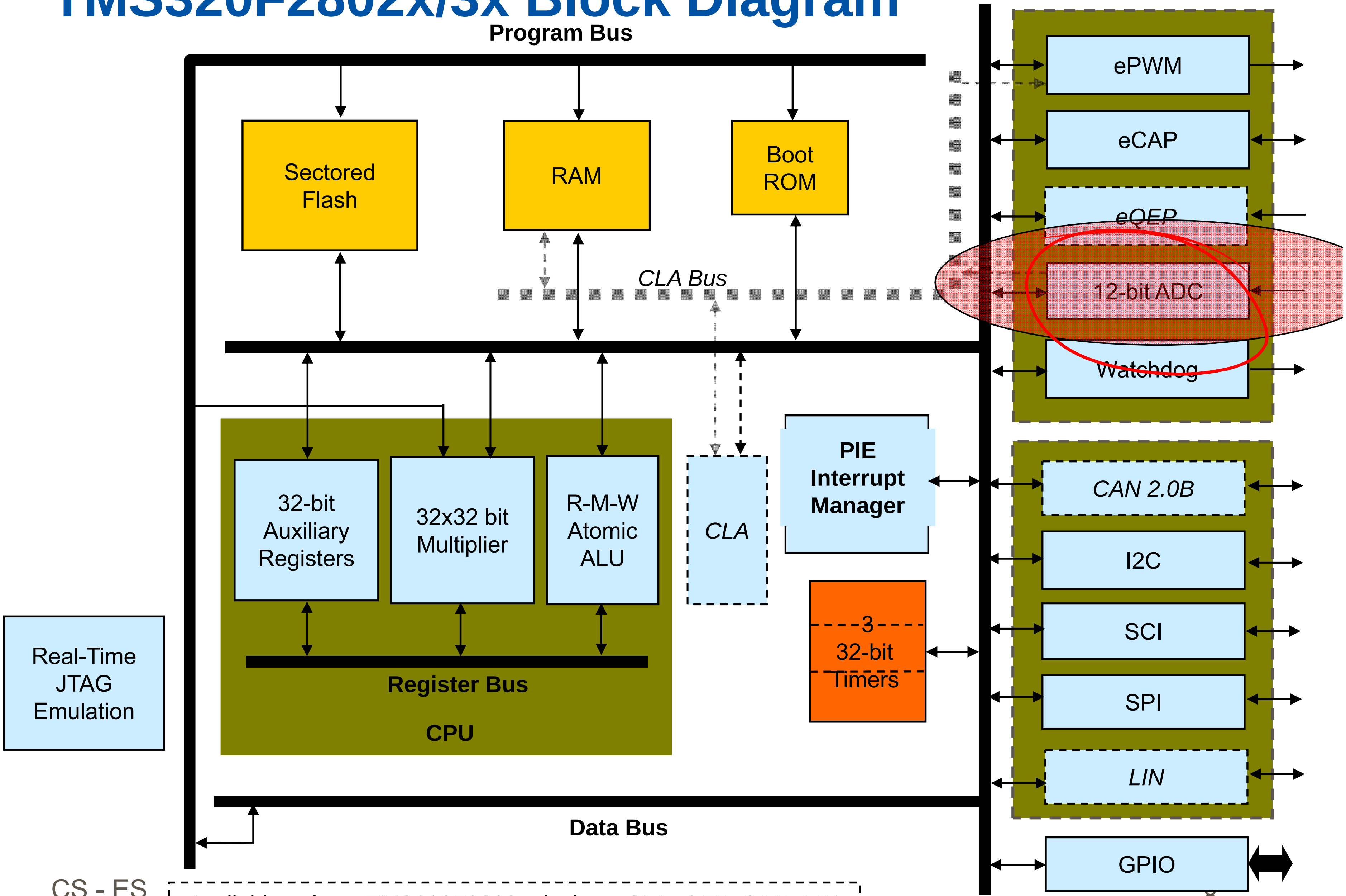
Automotive Radar, Electric Power Steering & Digital Power

Power Line Communications

LED Lighting

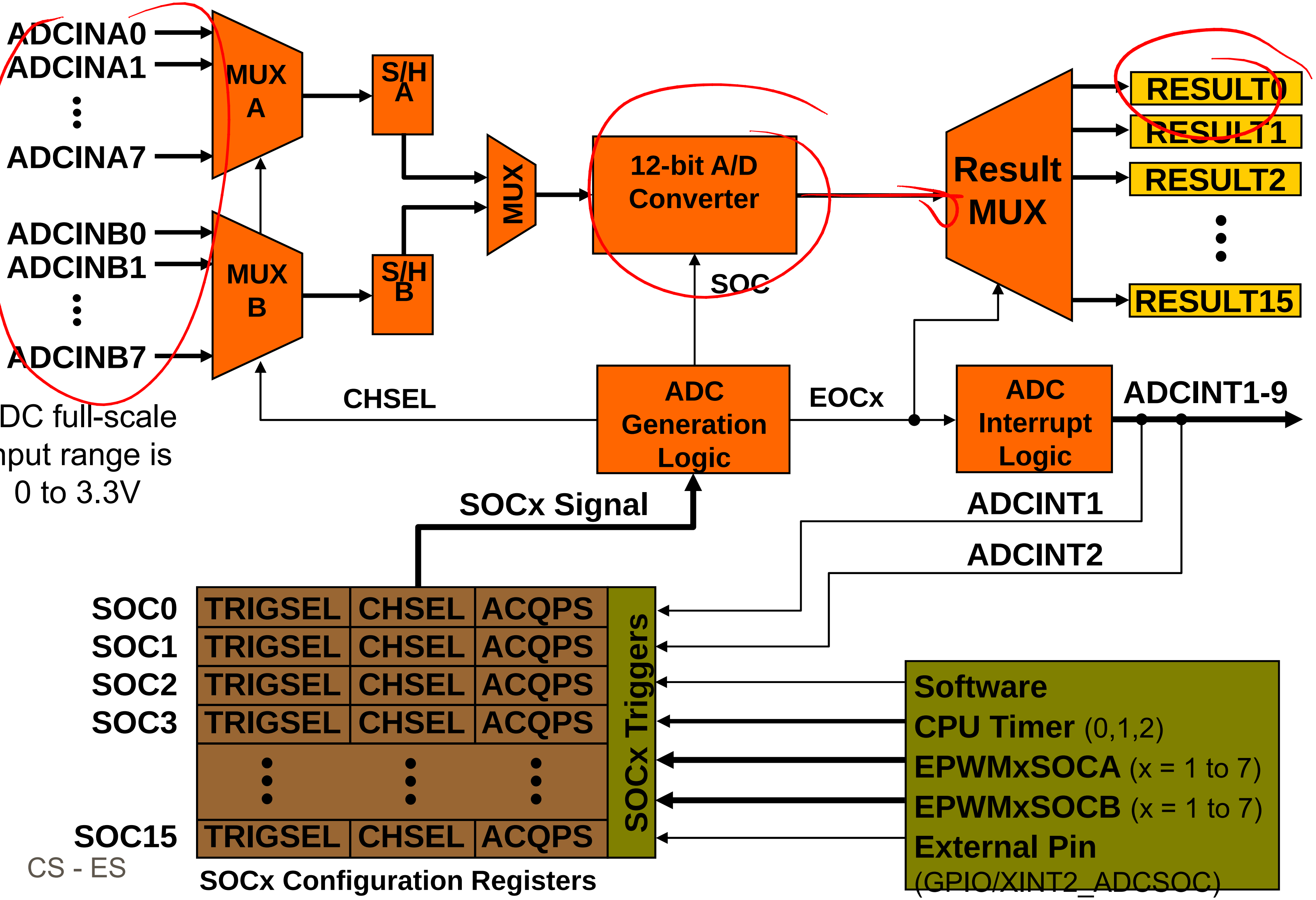
Consumer, Medical & Non-traditional

TMS320F2802x/3x Block Diagram



Available only on TMS320F2803x devices: CLA, QEP, CAN, LIN

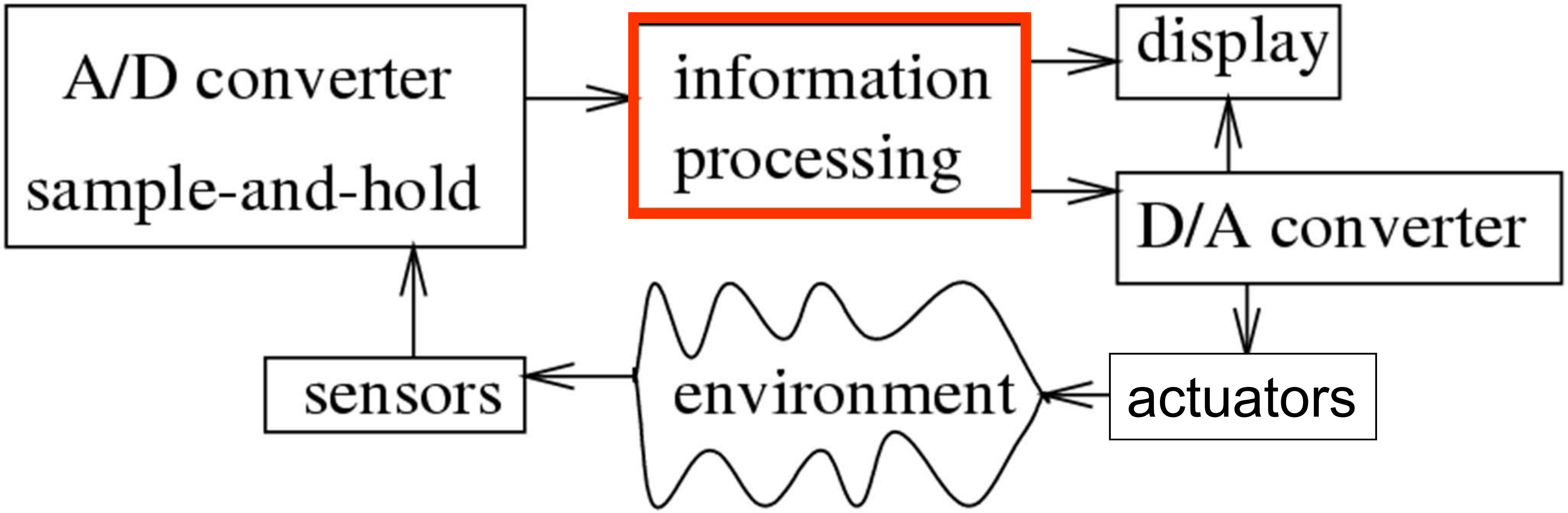
ADC Module Block Diagram



ADC full-scale input range is 0 to 3.3V

Embedded System Hardware

- Embedded system hardware is frequently used in a loop („***hardware in a loop***“):

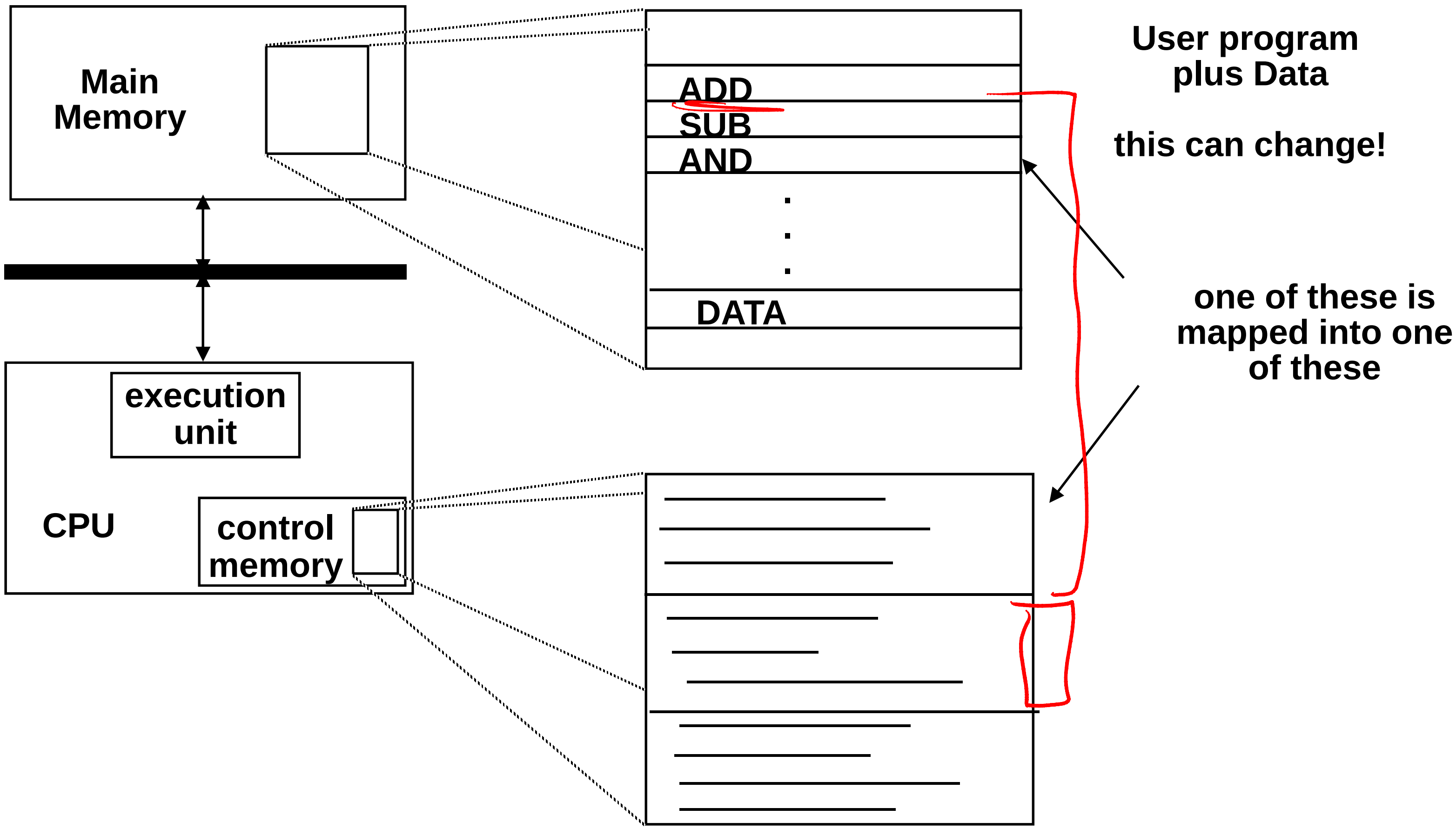


CISC vs. RISC

At the time of their initial development, CISC machines used available technologies to optimize computer performance.

- Microprogramming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit.
- The ease of microcoding new instructions allowed designers to make CISC machines upwardly compatible: a new computer could run the same programs as earlier computers because the new computer would contain a superset of the instructions of the earlier computers.
- Because microprogram instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated.

Microprogramming



Supported complex instructions a sequence of simple micro-inst

What is RISC?

- RISC, or *Reduced Instruction Set Computer*. is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.

- About 80% of the computations of a typical program required only about 20% of the instructions in a processor's instruction set. The most frequently used instructions were simple instructions such as load, store and add.

- Certain design features have been characteristic of most RISC processors:
  - ***one cycle execution time***: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called PIPELINING
  - ***pipelining*****: a techique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;**
  - ***large number of registers:*** the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC's disadvantages

- **Code Quality**

  The performance of a RISC processor depends greatly on the code that it is executing.

  If the programmer (or compiler) does a poor job of instruction scheduling, the processor can spend quite a bit stalling: waiting for the result of one instruction before it can proceed with a subsequent instruction.

  Since the scheduling rules can be complicated, most programmers use a high level language (such as C or C++) and leave the instruction scheduling to the compiler.

  This makes the performance of a RISC application depend critically on the quality of the code generated by the compiler. Therefore, developers (and development tool suppliers such as Apple) have to choose their compiler carefully based on the quality of the generated code.

Comparision

Feature	RISC	CISC
Power	One or two mill watts	Many watts
Compute Speed	Up to a mega-flop	Up to several mega-flop
I/O	Custom, any sort of hardware	PC based options via a BIOS
Cost	Dollars	Tens to hundreds of Dollars
Environmental	High Temp, Low EM Emissions	Needs Fans
Operating System Port	Difficult - Roughly equivalent to making a Mac OS run on a SPARC Station	Load and Go- simplified by an industry standard BIOS

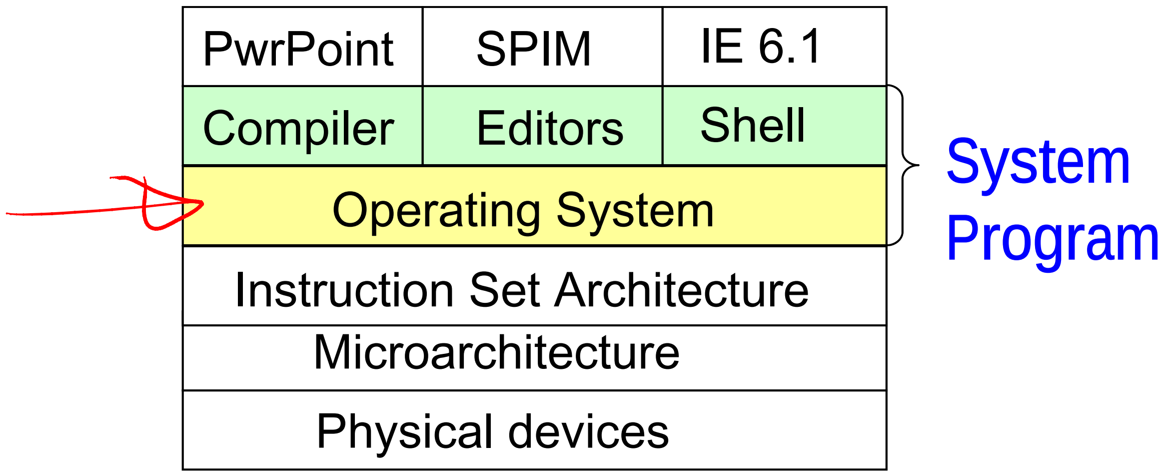
"Iron Law" of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology, and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology
- RISC systems shorten execution time by reducing the clock cycles per instruction.
- CISC systems improve performance by reducing the number of instructions per program.

What is an Operating System?

- An intermediate program between a user of a computer and the computer hardware (to hide messy details)
- Goals:
  - Execute user programs and make solving user problems easier
  - Make the computer system convenient and efficient to use



Operating System Concepts

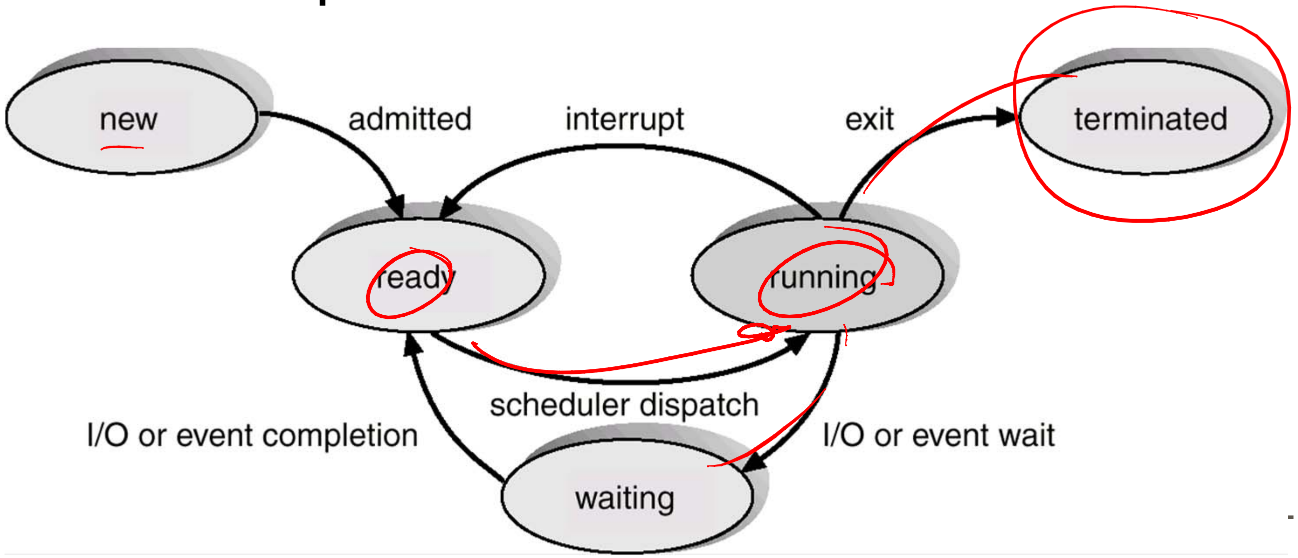
- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

Process Management

- A *process* is a program in execution
- A process contains
  - Address space (e.g. read-only code, global data, heap, stack, etc)
  - PC, $sp
  - Opened file handles
- A process needs certain resources, including CPU time, memory, files, and I/O devices
- The OS is responsible for the following activities for process management
  - Process creation and deletion
  - Process suspension and resumption
  - Provision of mechanisms for:
    - process synchronization
    - process communication

Process State

- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to be assigned to a process
  - running: Instructions are being executed
  - waiting: The process is waiting for some event (e.g. I/O) to occur
  - terminated: The process has finished execution

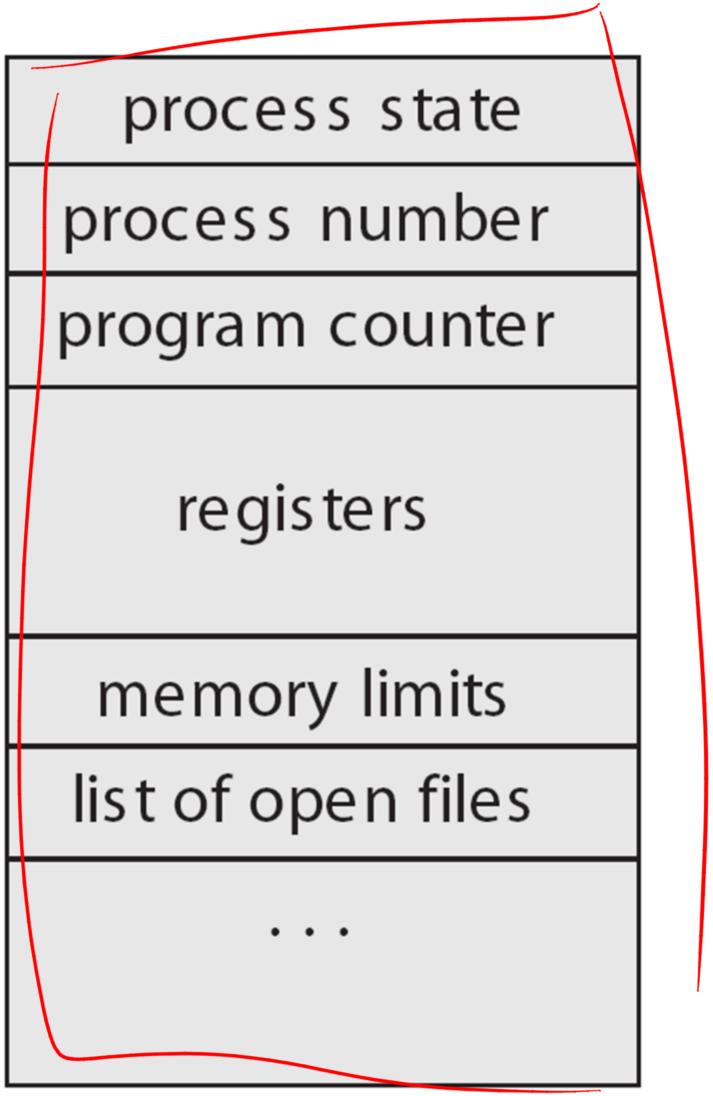


Process Control Block (PCB)

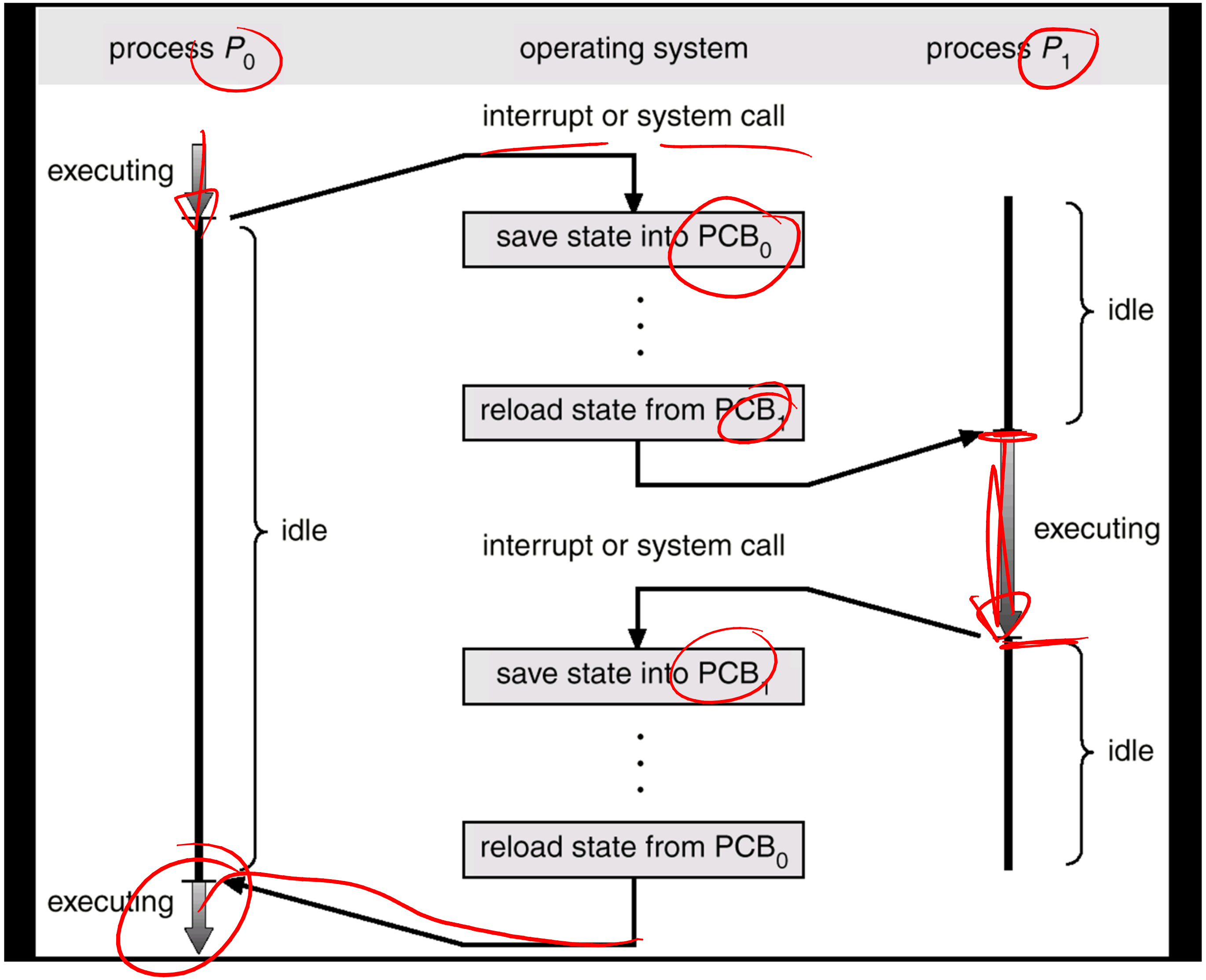
Information associated with each process

- Process state
- Program counter
- CPU registers (for context switch)
- CPU scheduling information (e.g. priority)
- Memory-management information (e.g. page table, segment table)
- Accounting information (PID, user time, constraint)
- I/O status information (list of I/O devices allocated, list of open files etc.)

Process Control Block (PCB)



CPU Switch From Process to Process



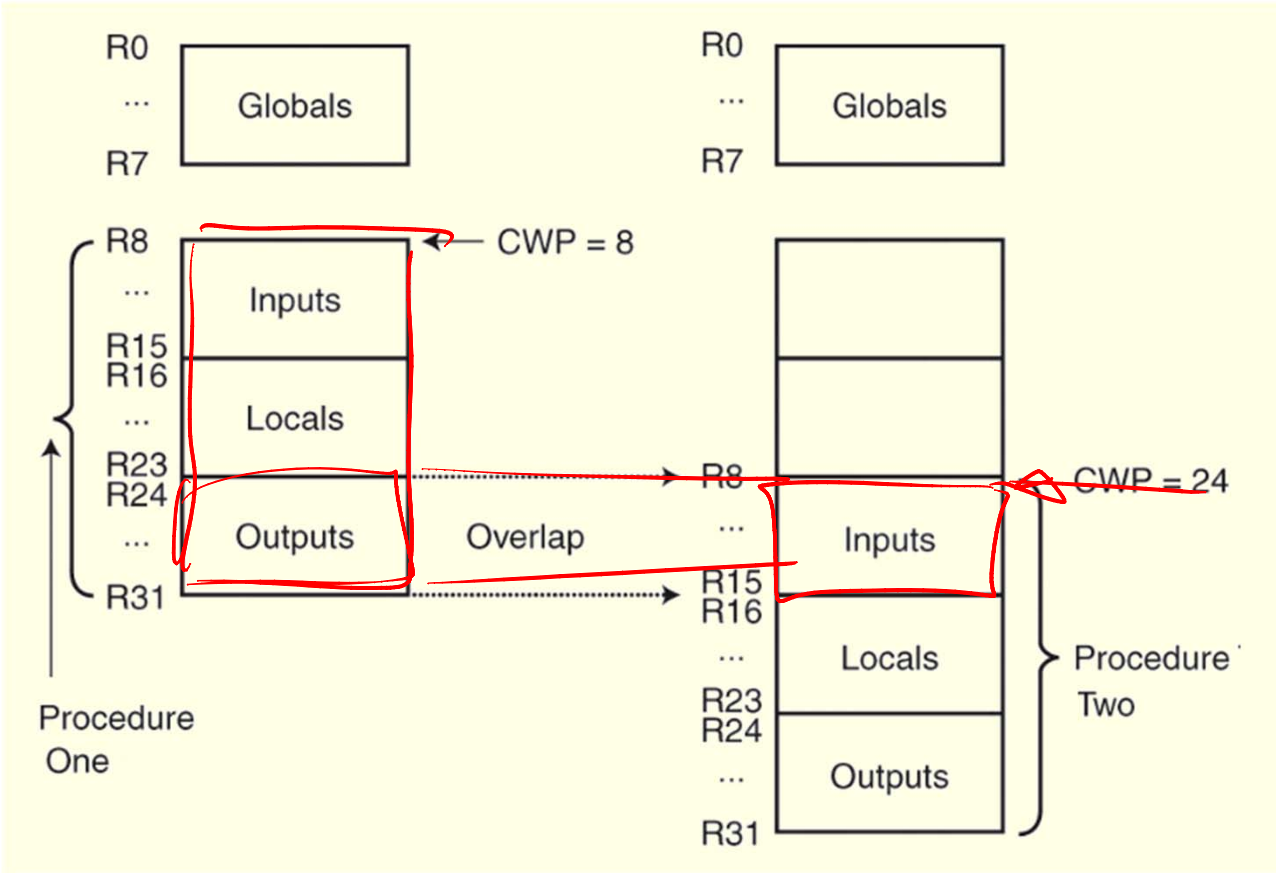
RISC Machines

- Because of their load-store ISAs, RISC architectures require a large numb of CPU registers.
- These register provide fast access to data during sequential program execution.
- They can also be employed to reduce the overhead typically caused by passing parameters to subprograms.
- Instead of pulling parameters off of a stack, the subprogram is directed to use a subset of registers.
- Fast Context Switching - support with two additional local register banks (e.g; Infineon XC167CI)
- E.g.; Berkeley RISC: > 100 Regs

  only 32 visible for the program.

RISC Machines

- This is how registers can be overlapped in a RISC system.
- The *current window pointer* (CWP) points to the active register window.

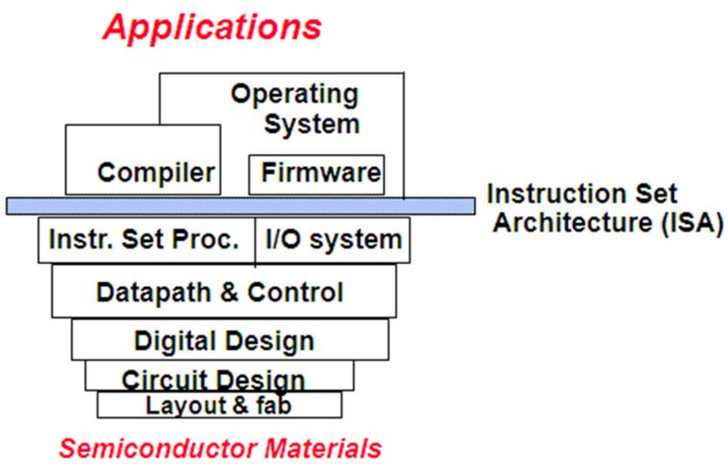


Instruction Set Architecture

REVIEW

Is the interface between hardware and software.

- allows easy programming (compilers, OS, ..);
  - Provides convenient functionality to higher levels
- allows efficient implementations (hardware);
  - Permits an efficient implementation at lower levels
- has a long lifetime (survives many HW generations) - portability



Instruction Set Architecture (ISA) versus Implementation

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: *MIPS, x86, IBM 360, JVM*
- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z990 (c. 2004)
  - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4 (c. 2000), AMD Athlon, Transmeta Crusoe, SoftPC*
  - MIPS implementations: *R2000, R4000, R10000, ...*
  - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*

Styles of ISA

- Accumulator
- Stack
- GPR

- CISC
- RISC
- VLIW
- Vector

- Boundaries are fuzzy, and hybrids are common
  - E.g., 8086/87 is hybrid accumulator-GPR-stack ISA
  - Many ISAs have added vector extensions

Infineon technologies

XC167
Derivatives

Preliminary

Functional Description

3 Functional Description

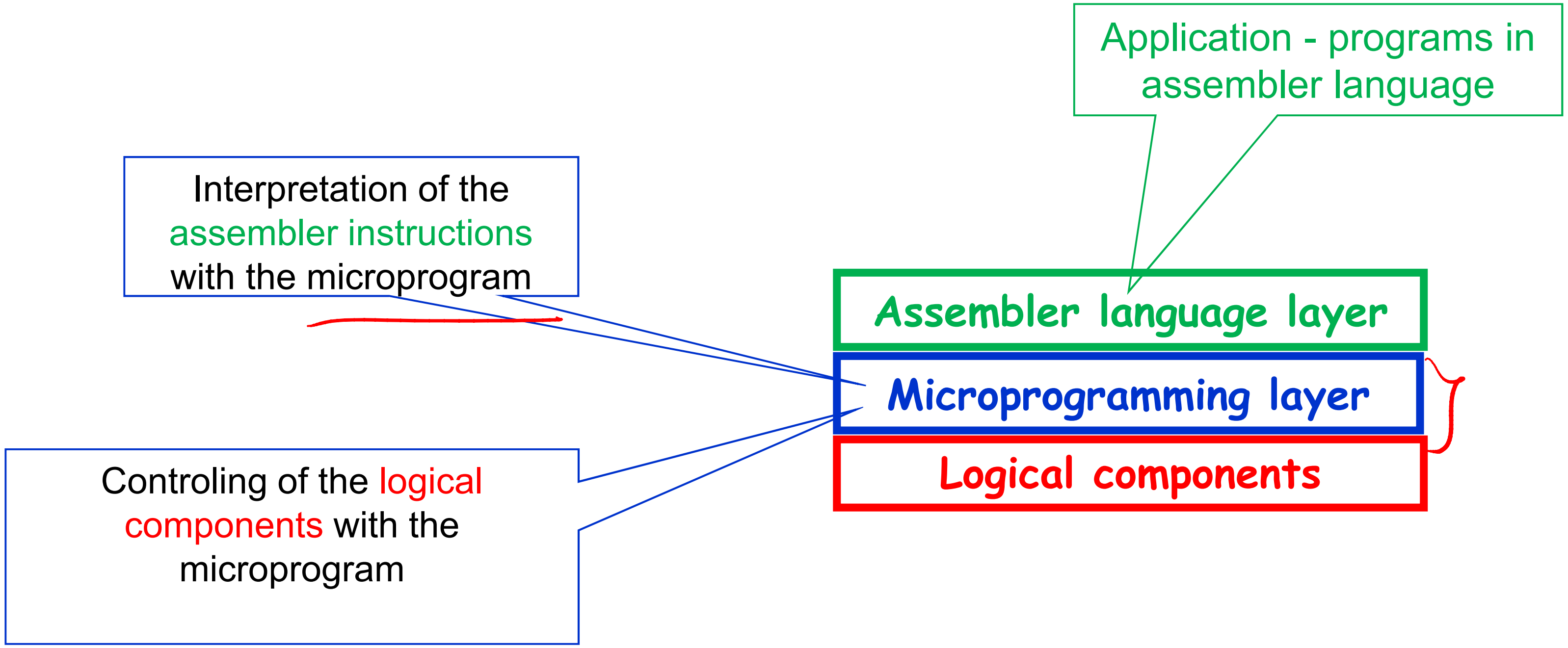
The architecture of the XC167 combines advantages of RISC, CISC, and DSP processors with an advanced peripheral subsystem in a very well-balanced way. In addition, the on-chip memory blocks allow the design of compact systems-on-silicon with maximum performance (computing, control, communication).

Styles of Implementation

- Microcoded
- Unpipelined single cycle
- Hardwired in-order pipeline
- Software interpreter
- Just-in-Time compiler

Micro programming

Tasks of the MP layer



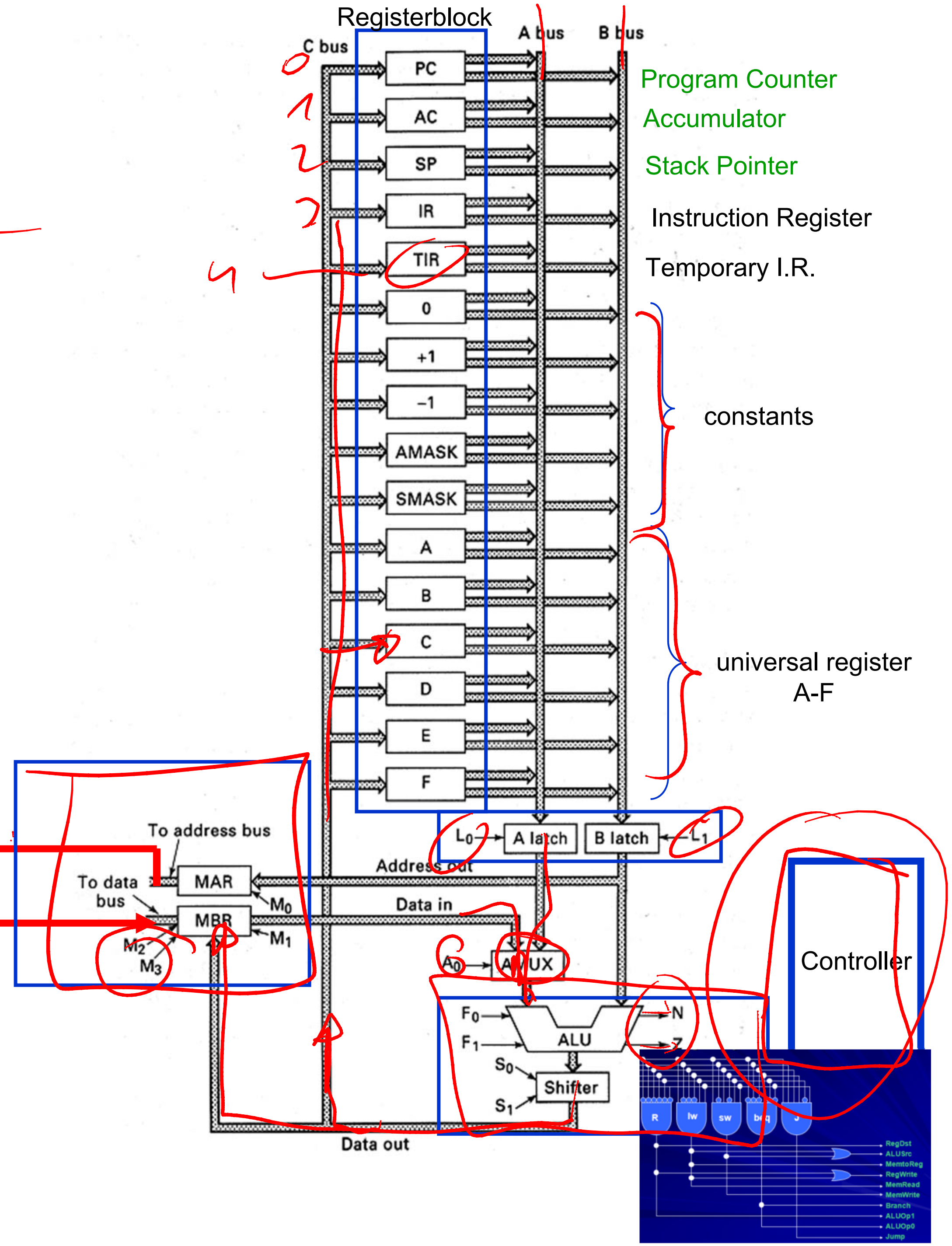
Microarchitecture

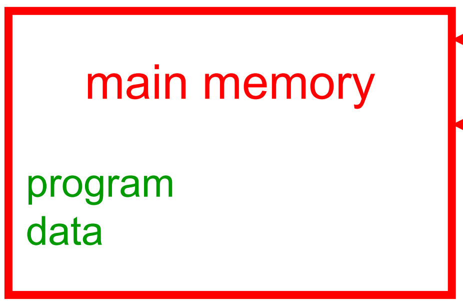
Registerbank
16   16-Bit register
special register: PC, AC, SP usw.
universalRegister: A-F

ALU
16-Bit
4 funktions (F0 , F1)
- A + B
- A and B
- $\bar{A}$
- A

2 statusbits (ALU-result)
- N   (neg.)
- Z   (zero)

shifter
1 Bit t



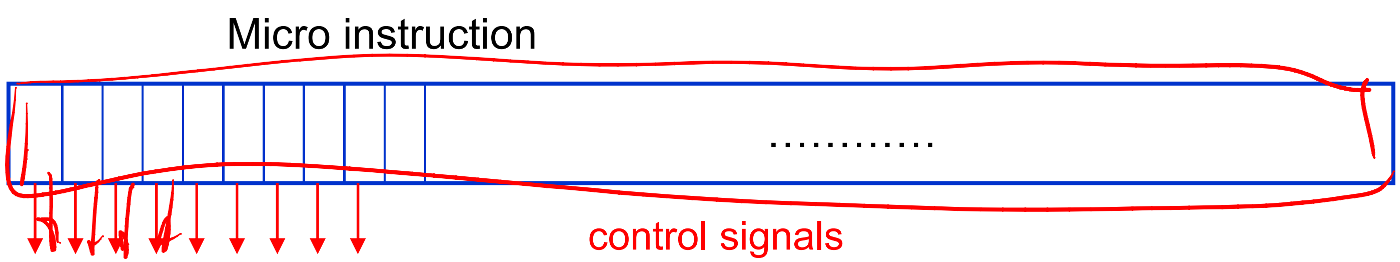


Format micro instruction

Signals for data path and memory:

16	control signals	load A-Bus
16	control signals	load B-Bus
16	-"-	load C-Bus
2	-"-	A, B- Latch
2	-"-	ALU-functions
2	-"-	shifter
1	-"-	MAR (M0)
3	-"-	MBR (M1), memory read/write (M2, M3)
1	-"-	AMUX (A0)
1	-"-	Enable C-Bus (ENC)

60 Bit per micro instruction

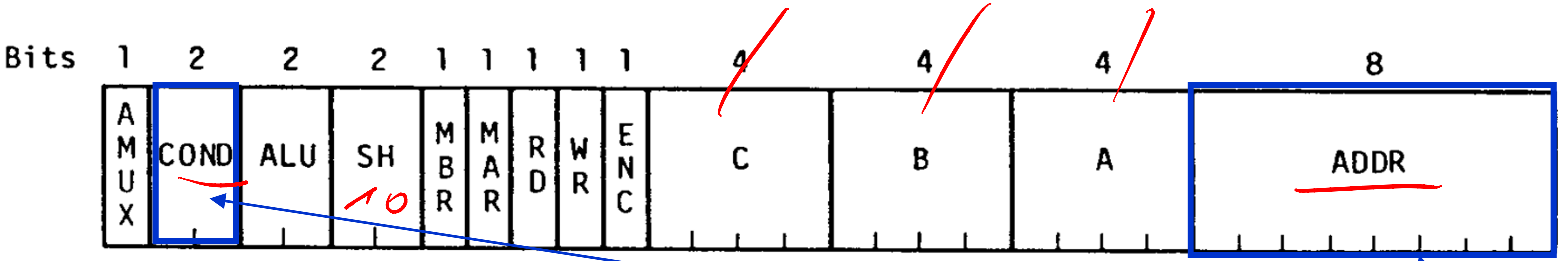


Format micro instruction

Reduction of the number of control bits

Use coding

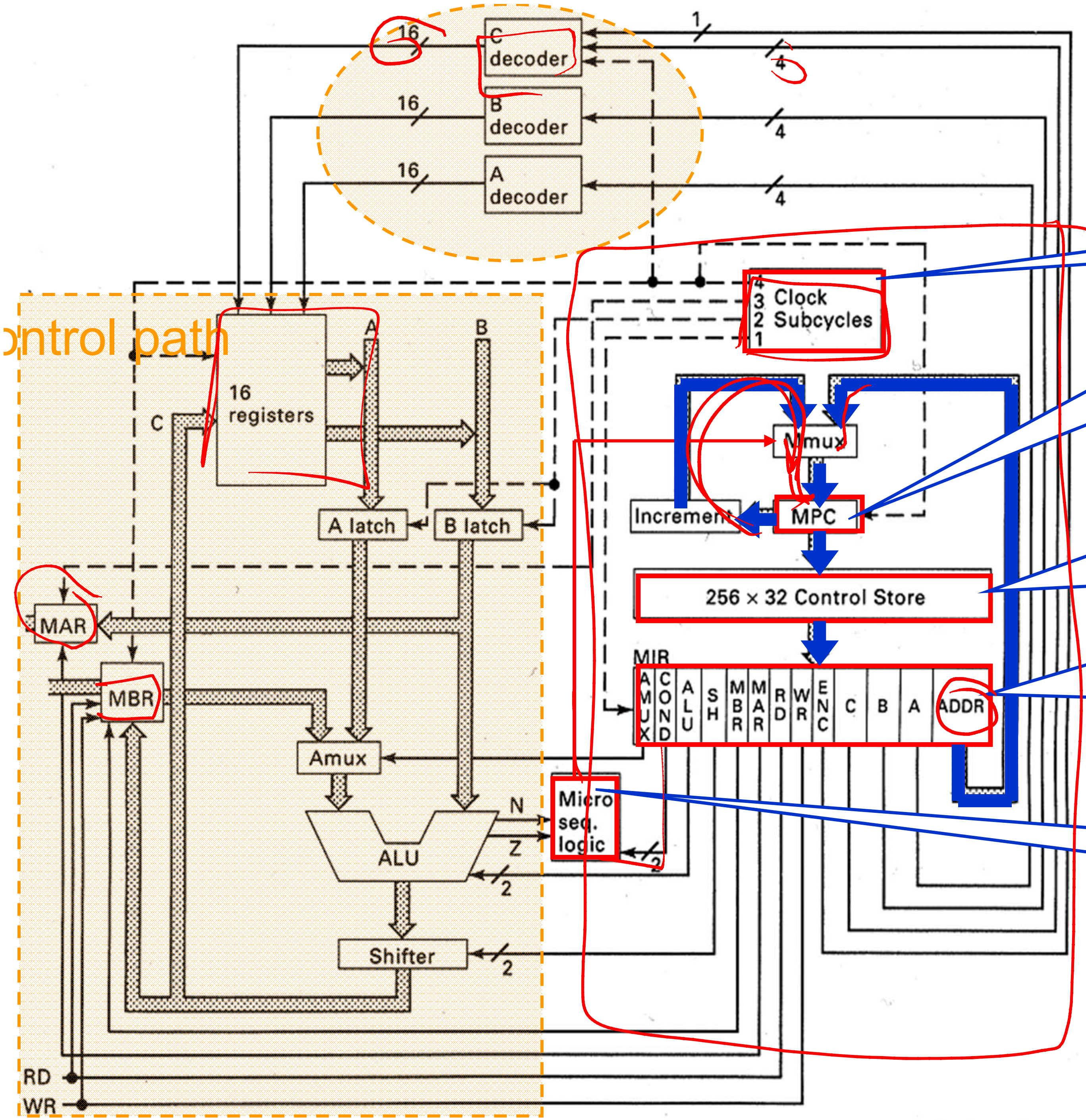
A-Bus 4 Bit (instead of 16)
B-Bus 4 Bit
C-Bus 4 Bit



⇨controll unit

AMUX	—controls left ALU input: 0 = A latch, 1 = MBR
ALU	—ALU function: 0 = A + B, 1 = A AND B, 2 = A, 3 = $\bar{A}$
SH	—shifter function: 0 = no shift, 1 = right, 2 = left
MBR	—load MBR from shifter: 0 = don't load MBR, 1 = load MBR
MAR	—load MAR from B latch: 0 = don't load MAR, 1 = load MAR
RD	—requests memory read: 0 = no read, 1 = load MBR from memory
WR	—requests memory write: 0 = no write, 1 = write MBR to memory
ENC	—controls storing into scratchpad: 0 = don't store, 1 = store
C	—selects register for storing into if ENC = 1: 0 = PC, 1 = AC, etc.
B	—selects B bus source: 0 = PC, 1 = AC, etc.
A	—selects A bus source: 0 = PC, 1 = AC, etc.

Microprogram control unit



Clock generator (4-phases)

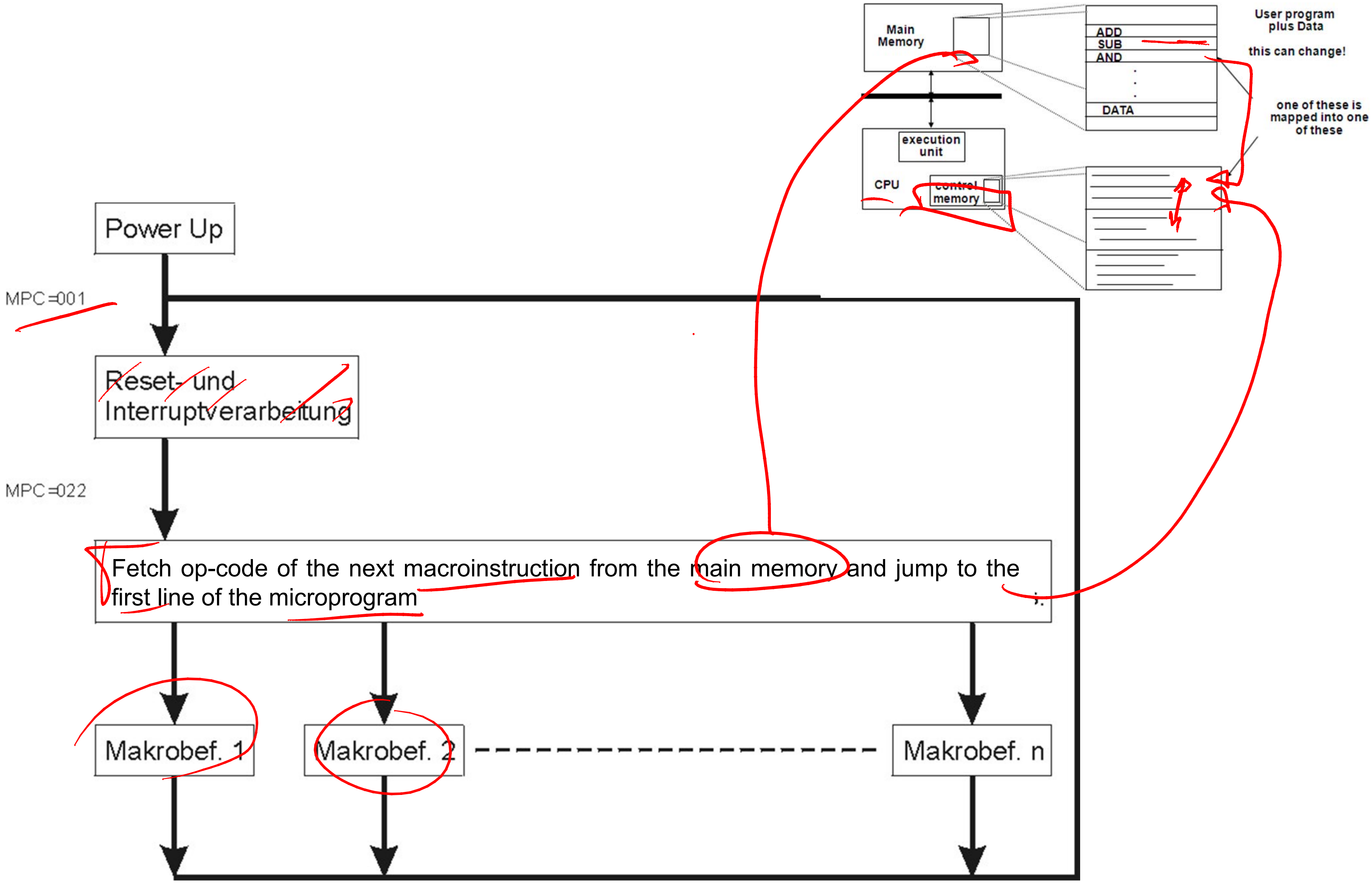
(MPC)

Microprogram memory
(256 words x 32 Bits):
Stores the micro program

Microinstruction register (MIR)

Microsequencer
(„next-Adrress-Logic)

Interpretation – macroinstruction



Microprogramm ("Interpreter") for the macroarchitecture

```
0: mar := pc ; rd ;                                   {main loop}
1: pc := pc + 1; rd ;                                 {increment pc}
2: ir := mbr ; if n then goto 28;                     {save, decode mbr}
3: tir := lshift (ir + ir ); if n then goto 19;
4: tir := lshift (tir ); if n then goto 11;           {000x or 001x?}
5: alu := tir ; if n then goto 9;                     {0000 or 0001?}
6: mar := ir ; rd ;                                   {0000 = LODD}
7: rd ;
8: ac := mbr ; goto 0;
9: mar := ir ; mbr := ac ; wr ;                       {0001 = STOD}
10: wr ; goto 0;
11: alu := tir ; if n then goto 15;                   {0010 or 0011?}
12: mar := ir ; rd ;                                  {0010 = ADDD}
13: rd ;
14: ac := mbr + ac ; goto 0;
15: mar := ir ; rd ;                                  {0011 = SUBD}
16: ac := ac + 1; rd ;                                {Note: x − y = x + 1 + not y}
17: a := inv (mbr );
18: ac := ac + a ; goto 0;
```

„000x“

- Fetch (0–1)
- Decode Opcode (Start) (2–5)
- Execute **LODD** (6–8)
- Execute **STOD** (9–10)
- Decode (2) (11)
- Execute **ADDD** (12–14)
- Execute **SUBD** (15–18)

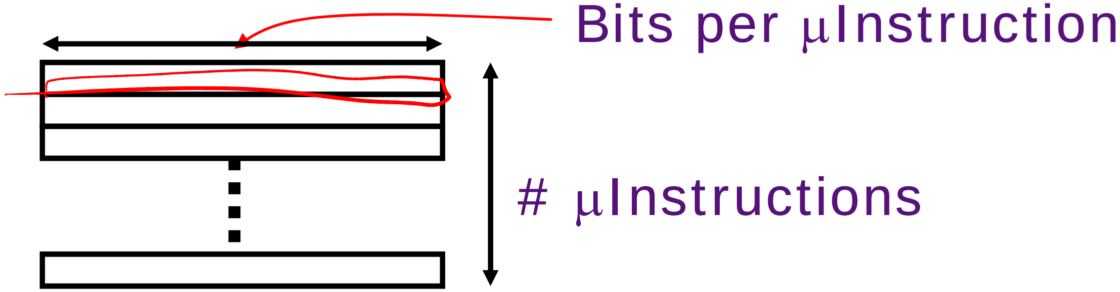
...

microinstruction

Register-Transfer-Notation

	AMUX	COND	ALU	SH	MBR	MAR	RD	WD	ENC	C	B	A	ADDR
mar := pc; rd;	0	0	2	0	0	1	1	0	0	0	0	0	00
rd;	0	0	2	0	0	0	1	0	0	0	0	0	00
ir := mbr	1	0	2	0	0	0	0	0	1	3	0	0	00
pc := pc + 1	0	0	0	0	0	0	0	0	1	0	6	0	00
mar := ir; mbr := ac; wr;	0	0	2	0	1	1	0	1	0	0	3	1	00
alu := tir; if *n* then goto 15;	0	1	2	0	0	0	0	0	0	0	0	4	15
ac := inv (mbr);	1	0	3	0	0	0	0	0	1	1	0	0	00
tir := lshift (tir); if *n* then goto 25;	0	1	2	2	0	0	0	0	1	4	0	4	25
alu := ac; if *z* then goto 22;	0	2	2	0	0	0	0	0	0	0	0	1	22
ac := band (ir, amask); goto 0	0	3	1	0	0	0	0	0	1	1	8	3	00
sp := sp + (−1); rd;	0	0	0	0	0	0	1	0	1	2	2	7	00
tir := lshift (ir + ir); if *n* then goto 69	0	1	0	2	0	0	0	0	1	4	3	3	69

Horizontal vs Vertical μCode



- Horizontal μcode has wider μinstructions
  - Multiple parallel operations per μinstruction
  - Fewer steps per macroinstruction
  - Sparser encoding ⇒ more bits
- Vertical μcode has narrower μinstructions
  - Typically a single datapath operation per μinstruction
    - separate μinstruction for branches
  - More steps to per macroinstruction
  - More compact ⇒ less bits
- Nanocoding
  - Tries to combine best of horizontal and vertical μcode

Dictionary approach, two level control store (indirect addressing of instructions)

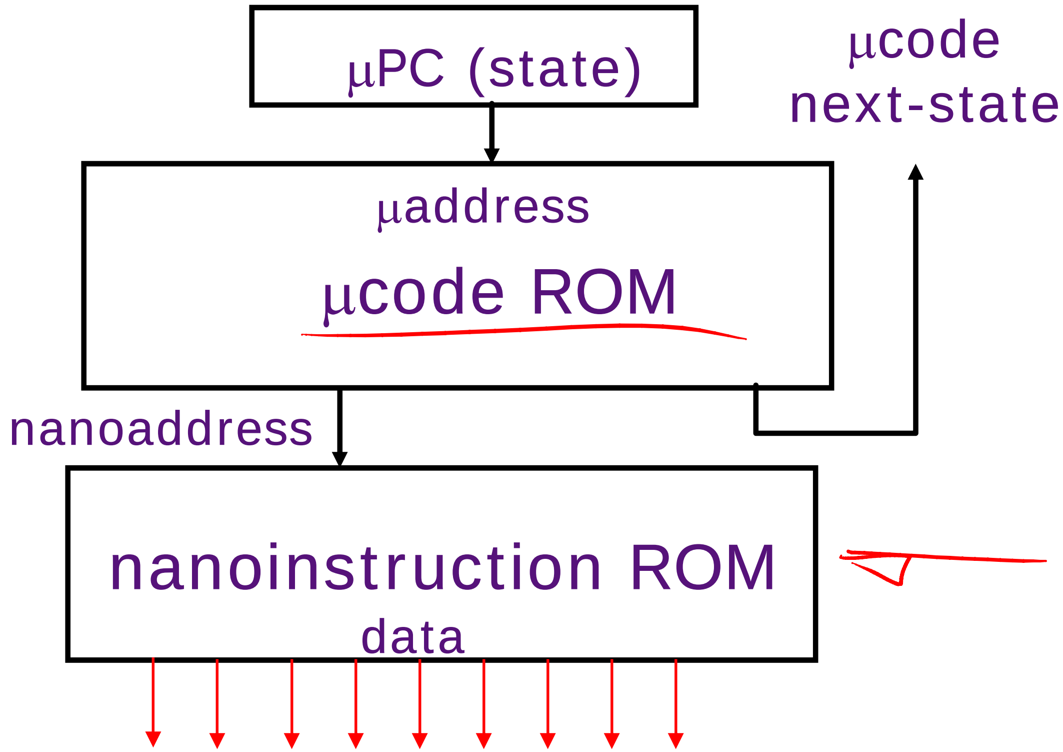
"*Dictionary-based coding schemes cover a wide range of various coders and compressors.*
Their common feature is that the methods use some kind of a dictionary that contains parts of the input sequence which frequently appear.
The encoded sequence in turn contains references to the dictionary elements rather than containing these over and over."

[Á. Beszédes et al.: Survey of Code size Reduction Methods, Survey of Code-Size Reduction Methods, *ACM Computing Surveys*, Vol. 35, Sept. 2003, pp 223-267]

Nanocoding

Exploits recurring control signal patterns in μcode, e.g.,

ALU_0 A ← Reg[rs]
...
$ALUi_0$ A ← Reg[rs]
...



- MC68000 had 17-bit μcode containing either 10-bit μjump or 9-bit nanoinstruction pointer
  - Nanoinstructions were 68 bits wide, decoded to give 196 control signals

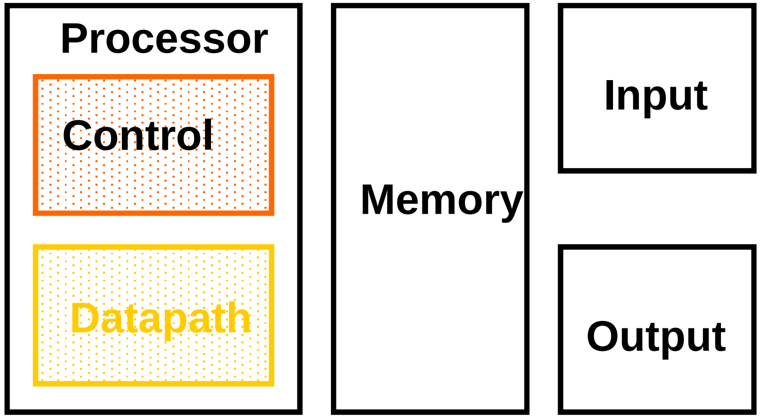
Microprogramming in Modern Usage

- *Microprogramming is far from extinct*
- Played a crucial role in micros of the Eighties
  DEC uVAX, Motorola 68K series, Intel 386 and 486
- Microcode pays an assisting role in most modern micros *(AMD Athlon, Intel Core 2 Duo, IBM PowerPC)*
  - Most instructions are executed directly, i.e., with hard-wired control
  - Infrequently-used and/or complicated instructions invoke the microcode engine
- *Patchable* microcode common for post-fabrication bug fixes, e.g. Intel Pentiums load µcode patches at bootup

Pipelining

Review: Single-cycle Processor

- Five steps to design a processor:
  1. Analyze instruction set → datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic
     - Formulate Logic Equations
     - Design Circuits



Single Cycle Performance

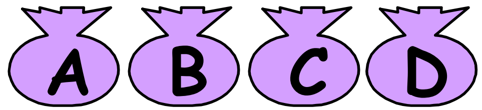
- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock rate is?

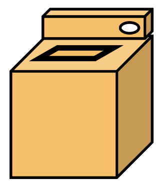
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

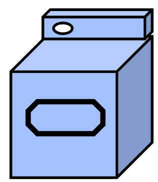
- What can we do to improve clock rate?
- Will this improve performance as well?

Pipelining: It's Natural!

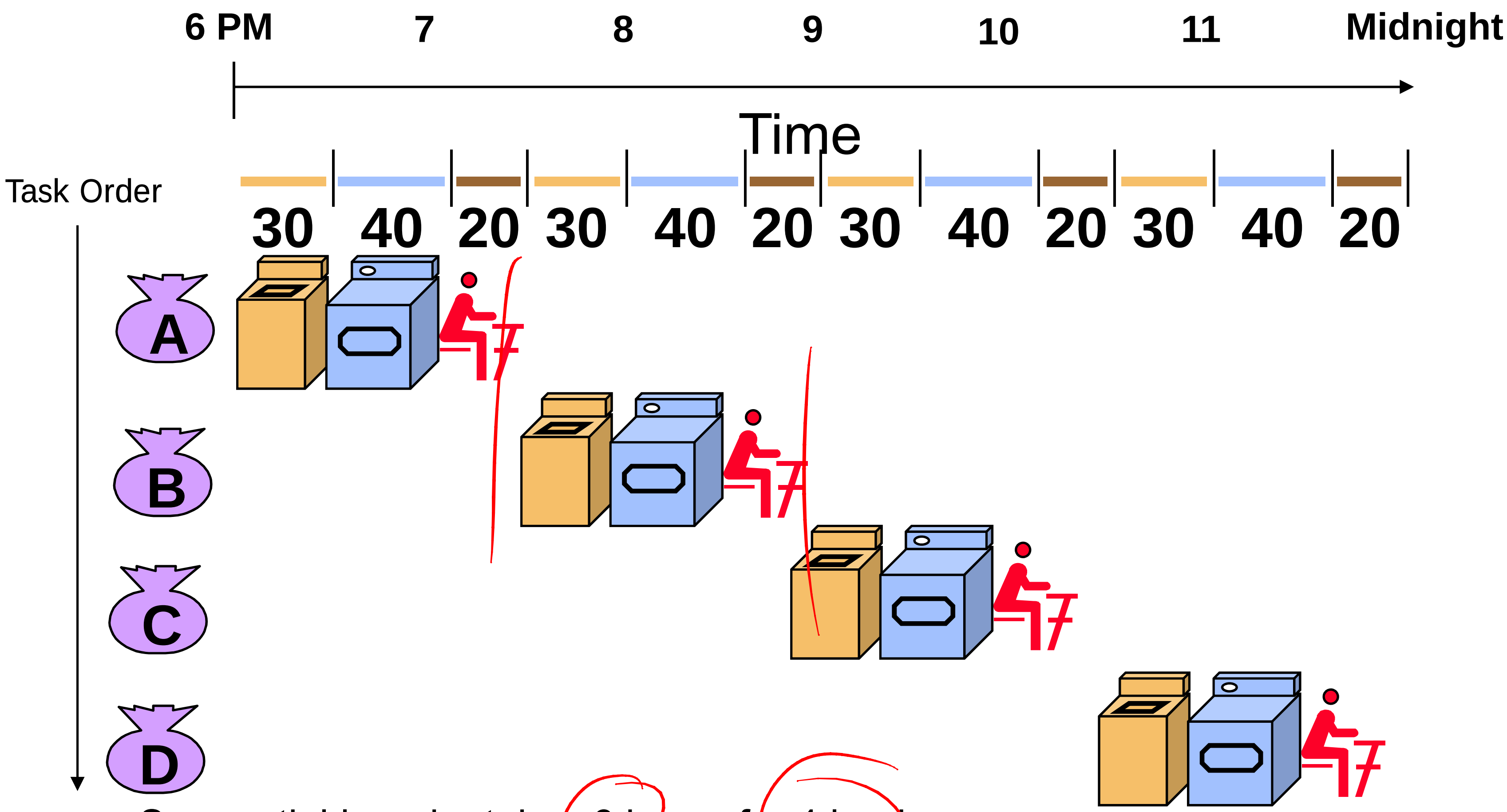
- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - "Folder" takes 20 minutes





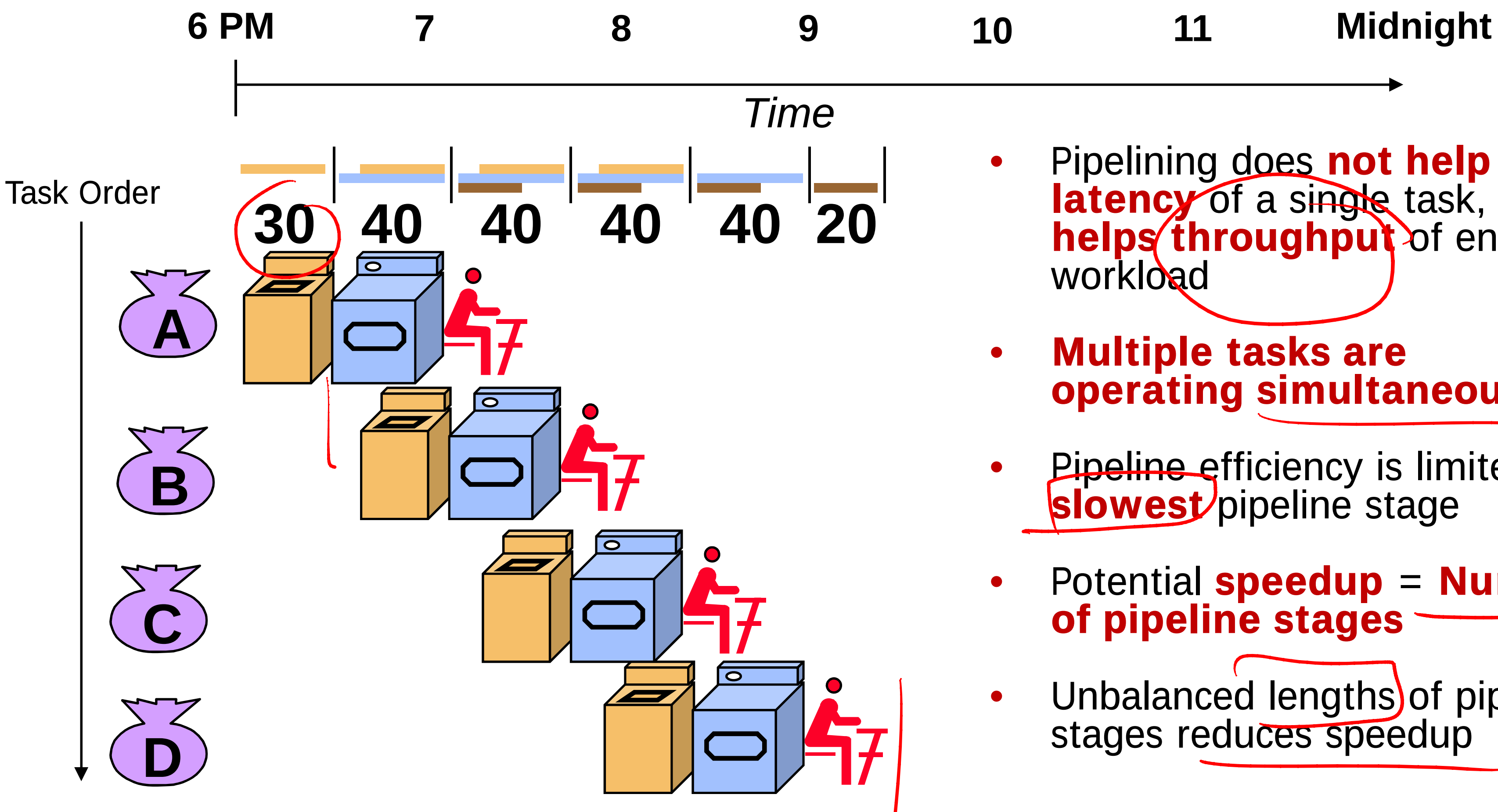


Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry: Why Wait?



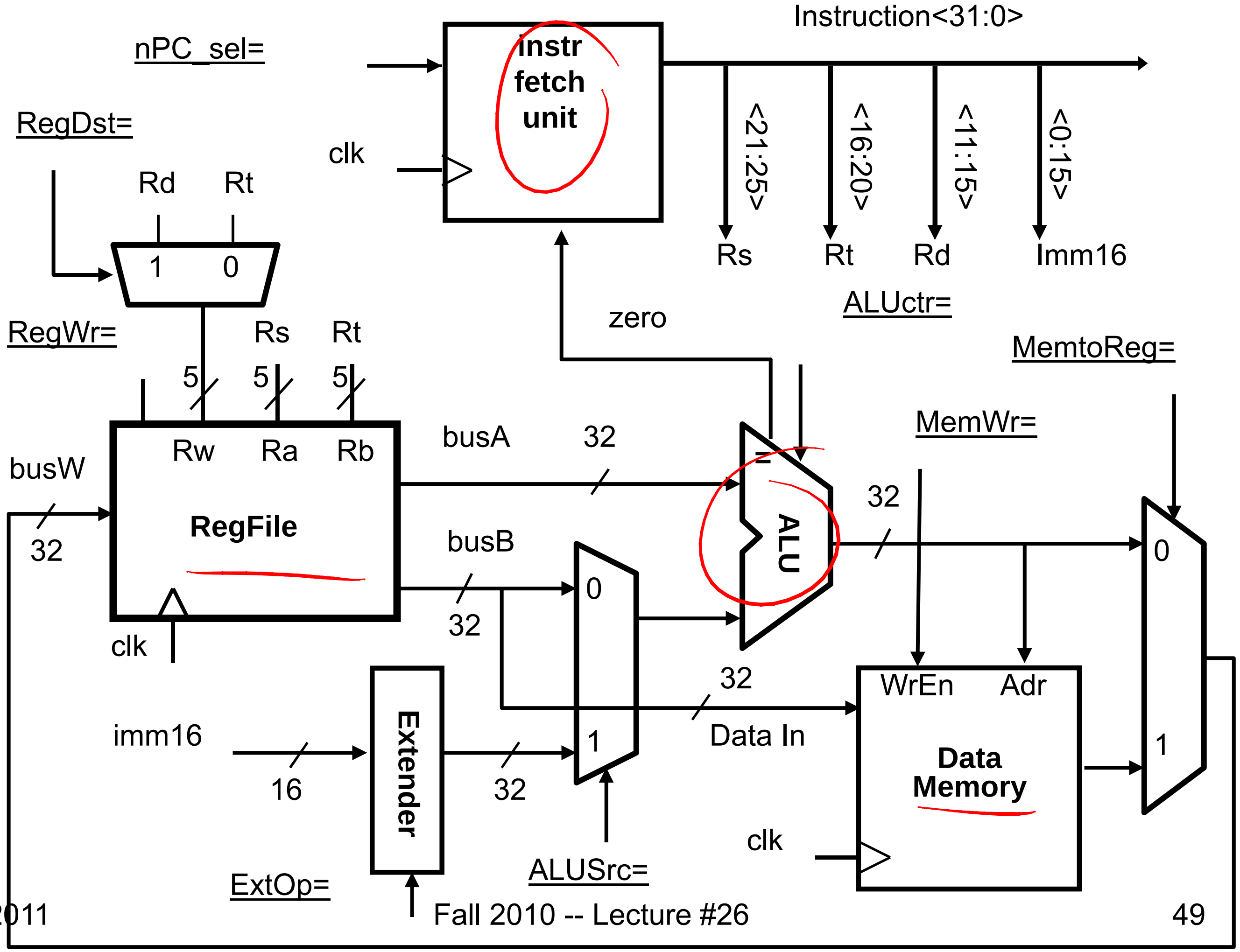
- Pipelining does **not help latency** of a single task, it **helps throughput** of entire workload
- **Multiple tasks are operating simultaneously**
- Pipeline efficiency is limited by **slowest** pipeline stage
- Potential **speedup** = **Number of pipeline stages**
- Unbalanced lengths of pipe stages reduces speedup

- Pipelined laundry takes 3.5 hours for 4 loads

Single Cycle Datapath

31	26	21	16	0
op	rs	rt	immediate	

- Data Memory {R[rs] + SignExt[imm16]} = R[rt]



Steps in Executing MIPS

1) IFtch: Instruction Fetch, Increment PC

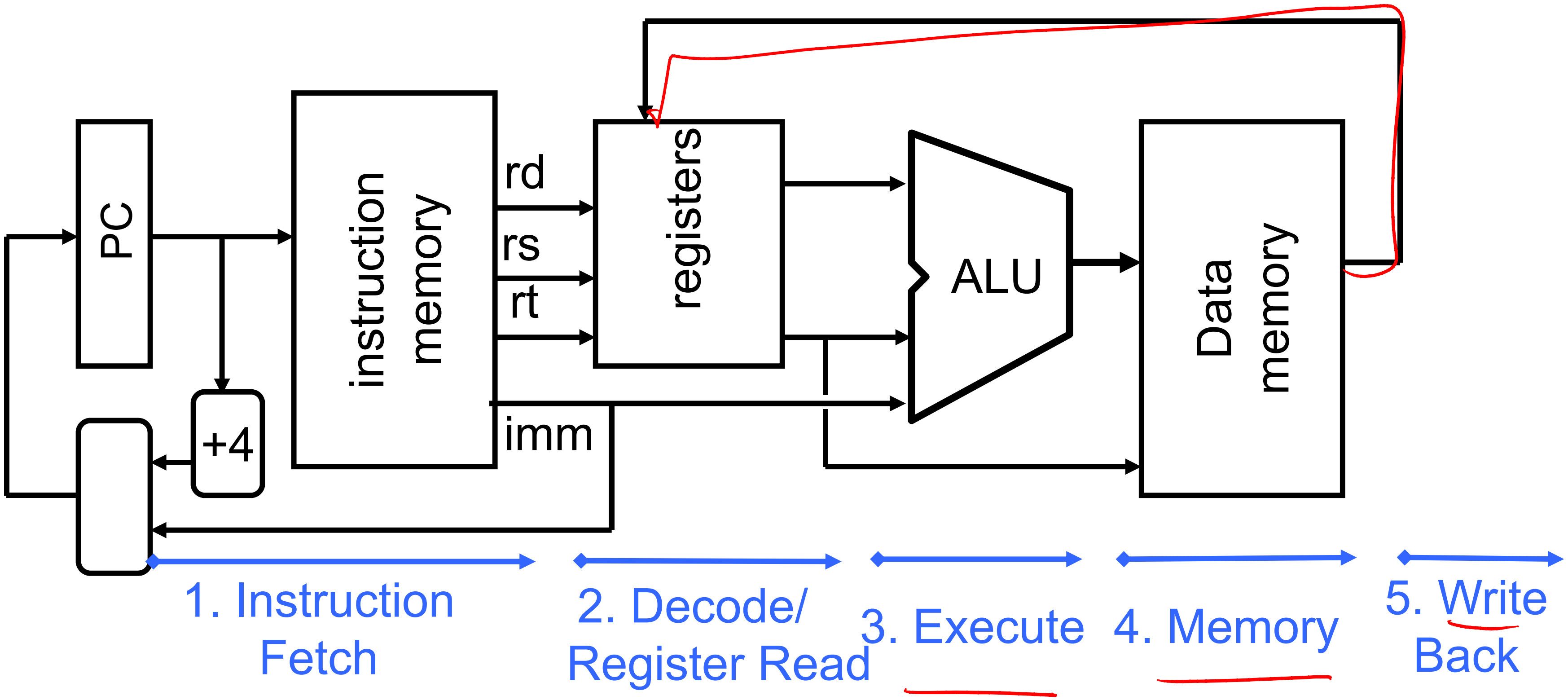
2) Dcd: Instruction Decode, Read Registers

3) Exec:
   Mem-ref: Calculate Address
   Arith-log: Perform Operation

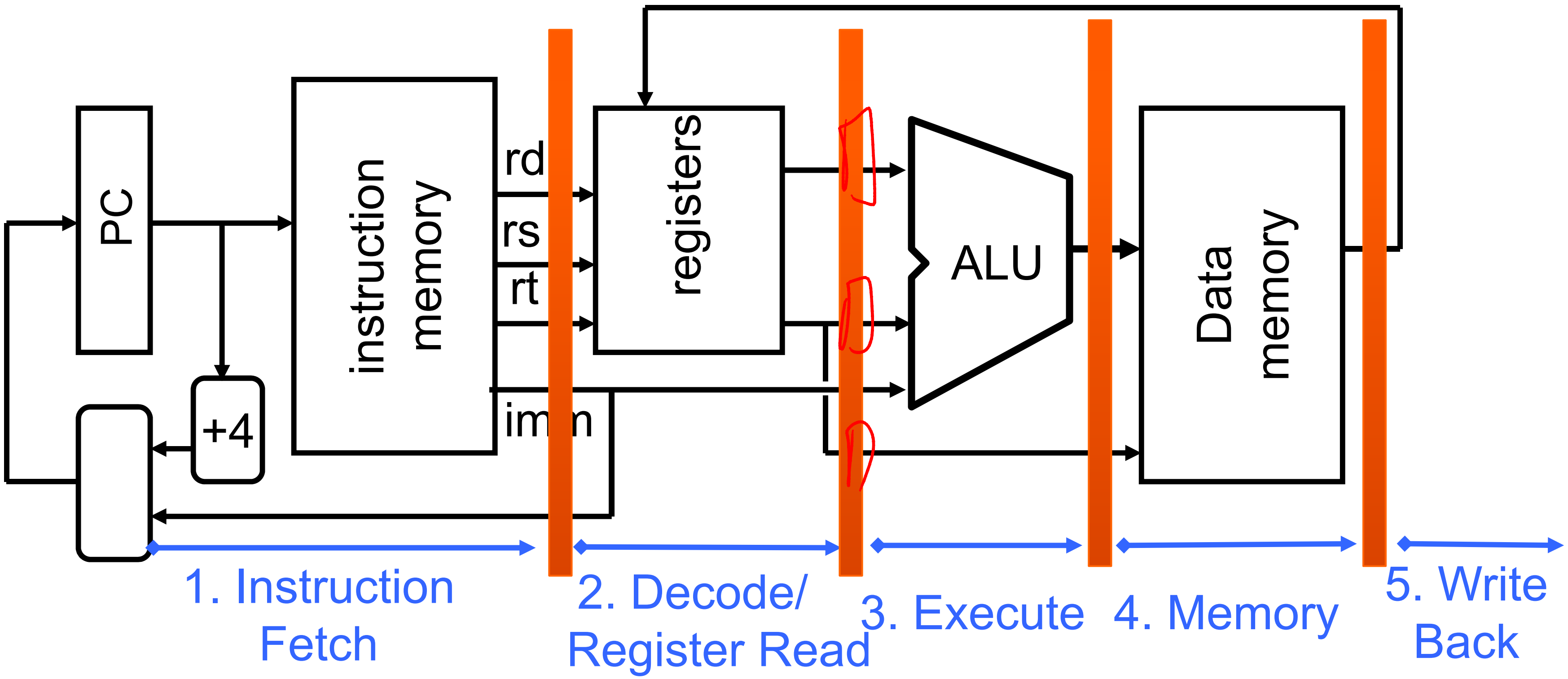
4) Mem:
   Load: Read Data from Memory
   Store: Write Data to Memory

5) WB: Write Data Back to Register

Redrawn Single Cycle Datapath

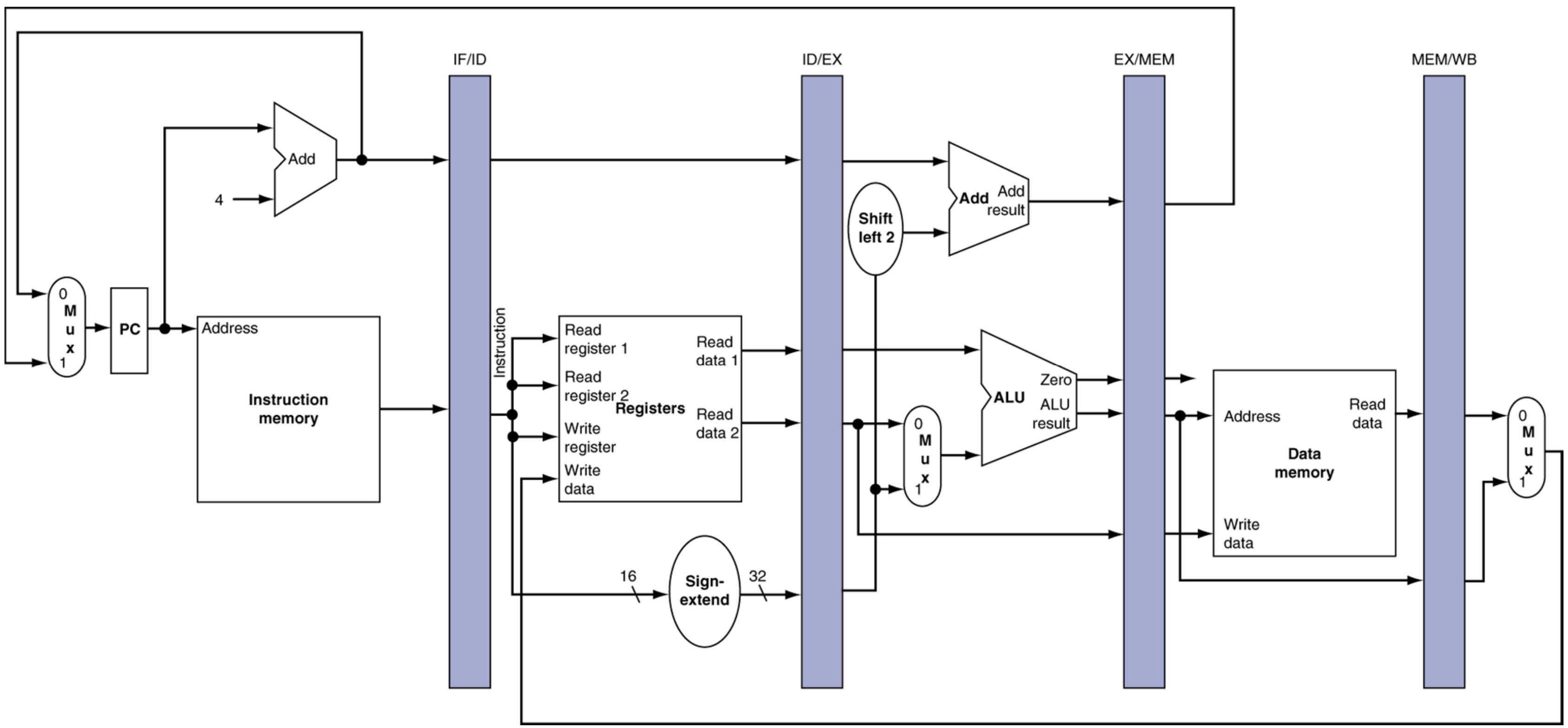


Pipeline registers

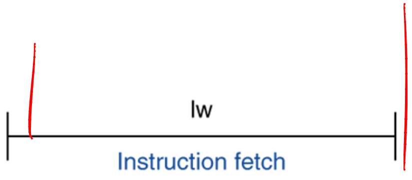


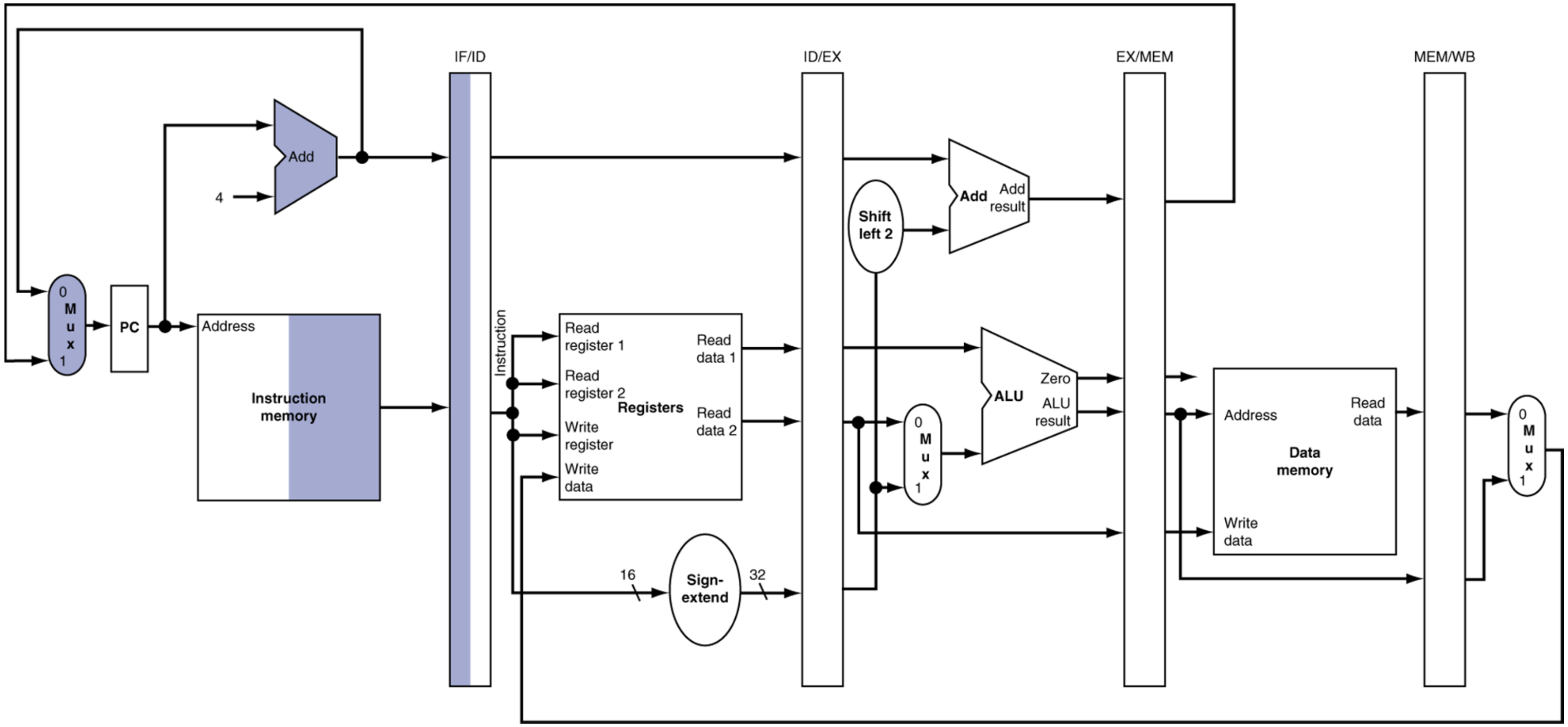
- Need registers between stages
  - To hold information produced in previous cycle

More Detailed Pipeline

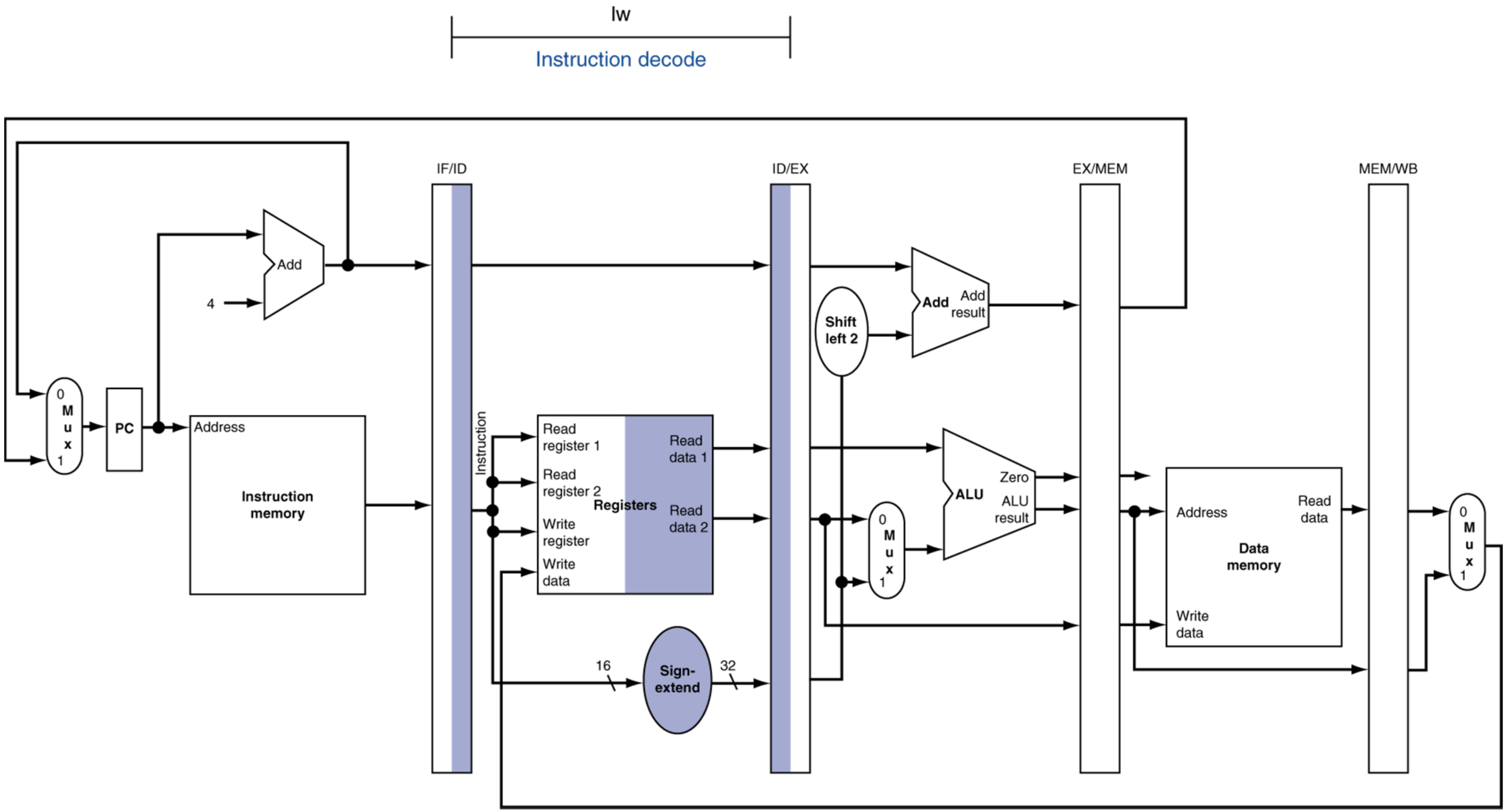


IF for Load, Store, …

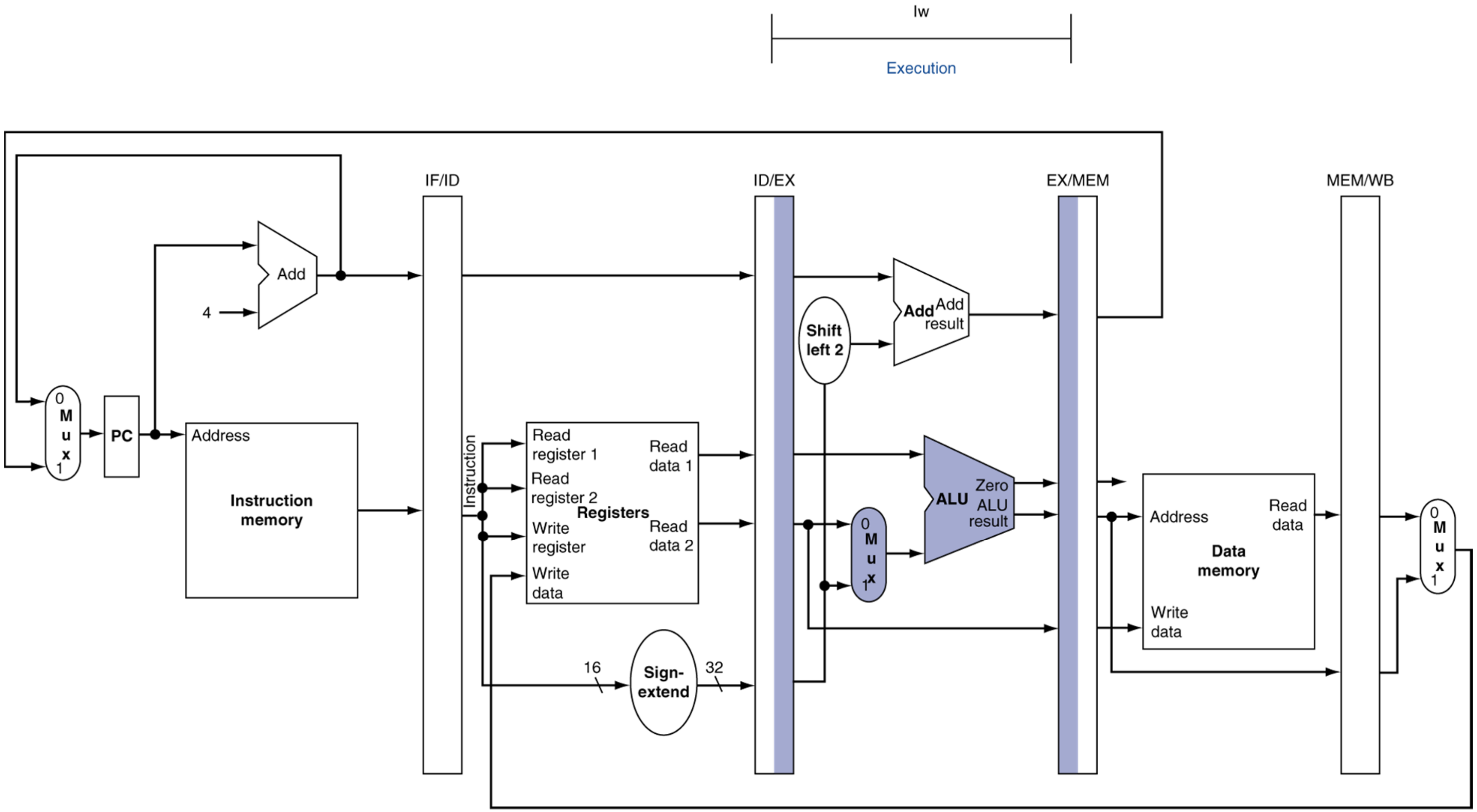




ID for Load, Store, …

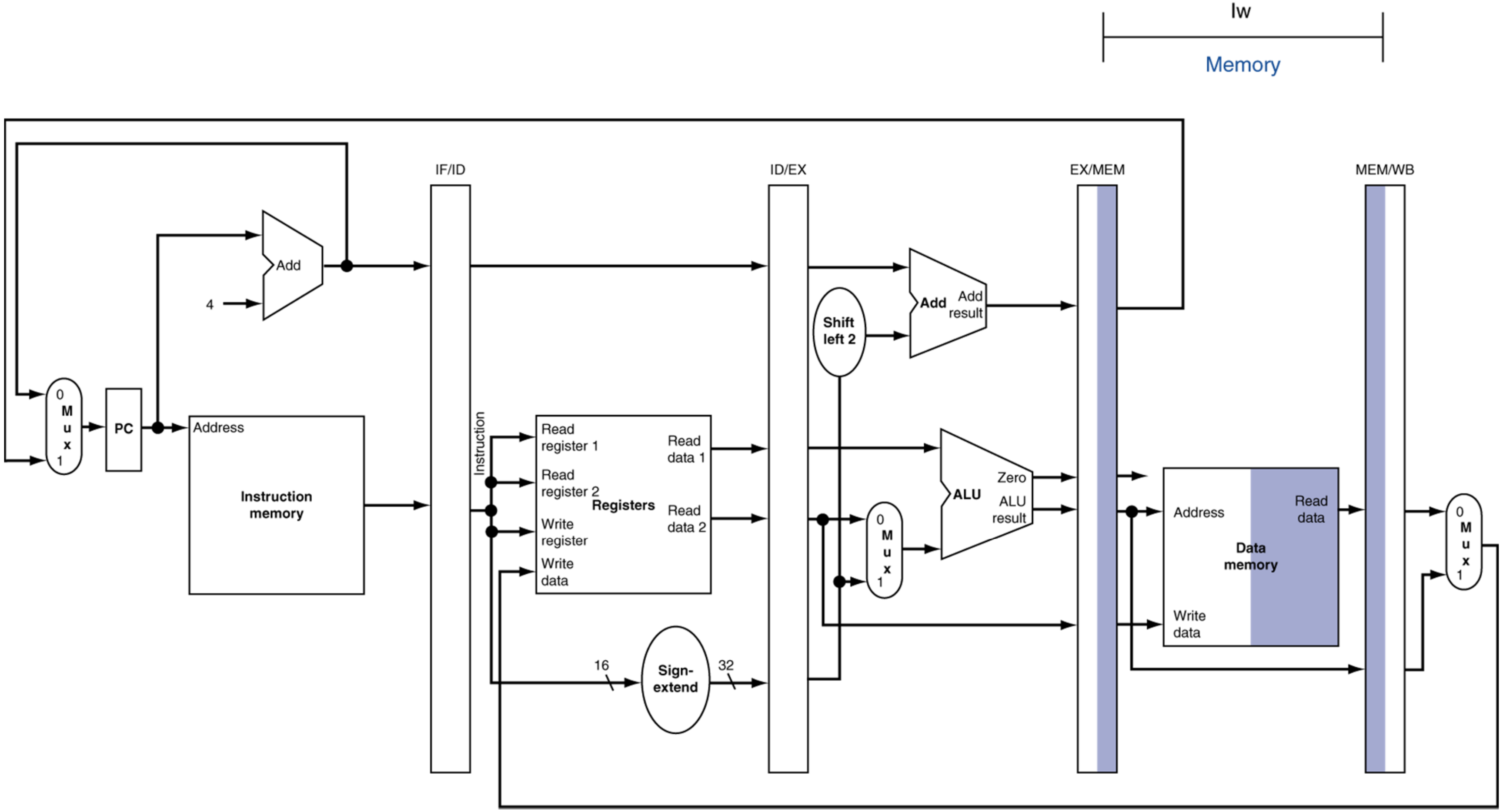


EX for Load



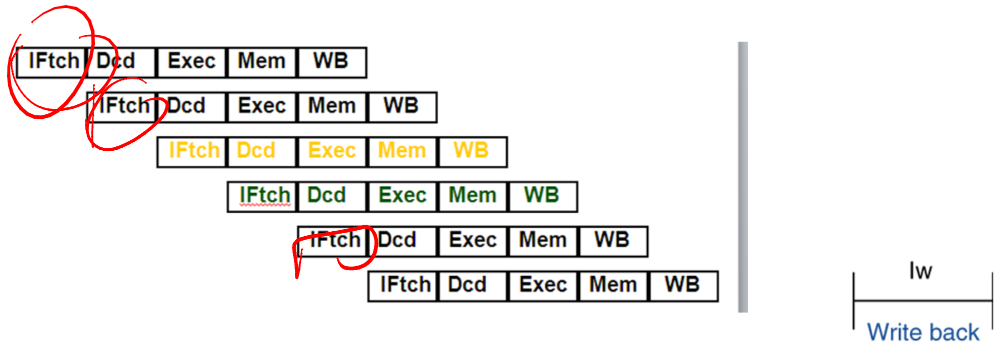
lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
0
M
u
x
1
ALU
Zero
ALU result
Address
Data memory
Write data
Read data
0
M
u
x
1

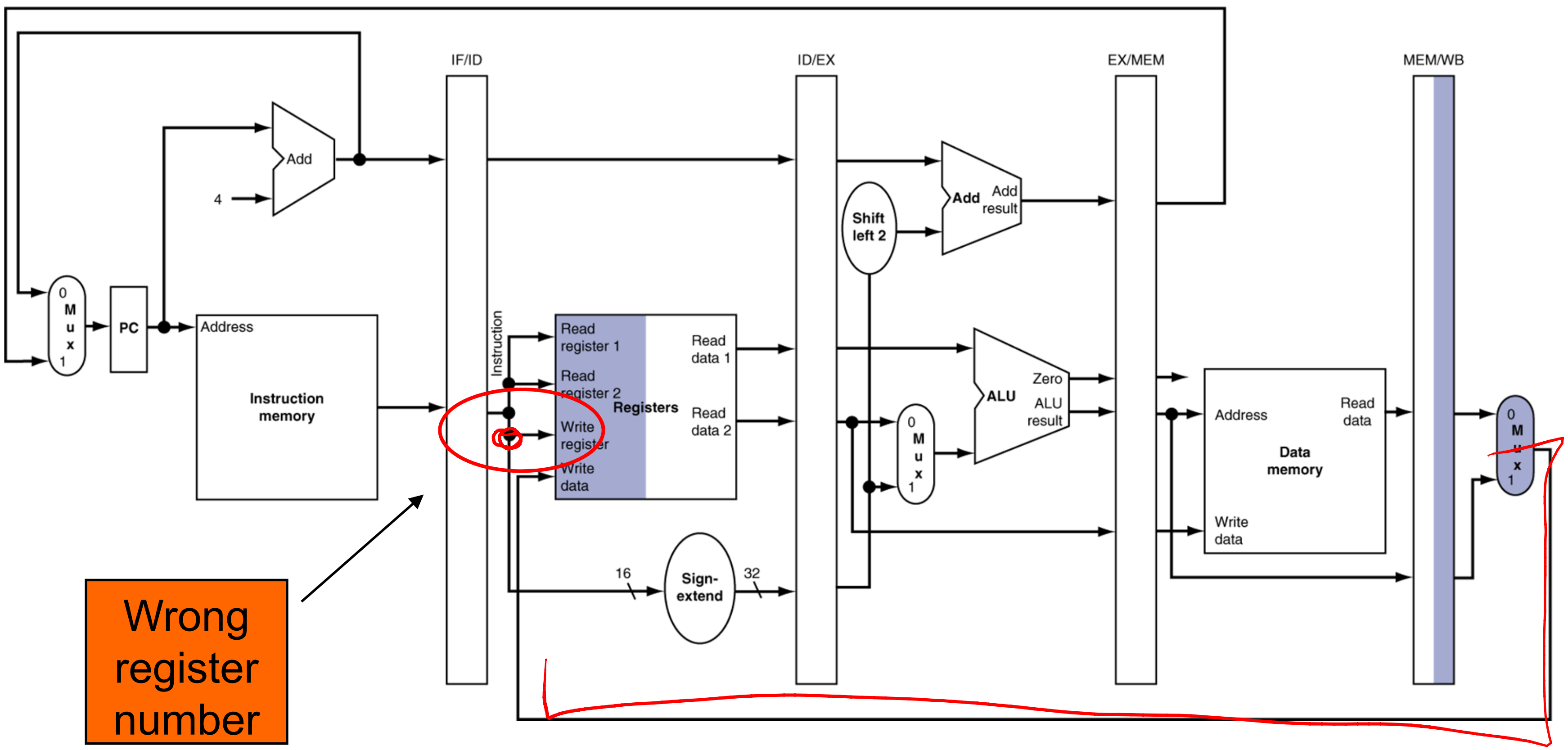
MEM for Load



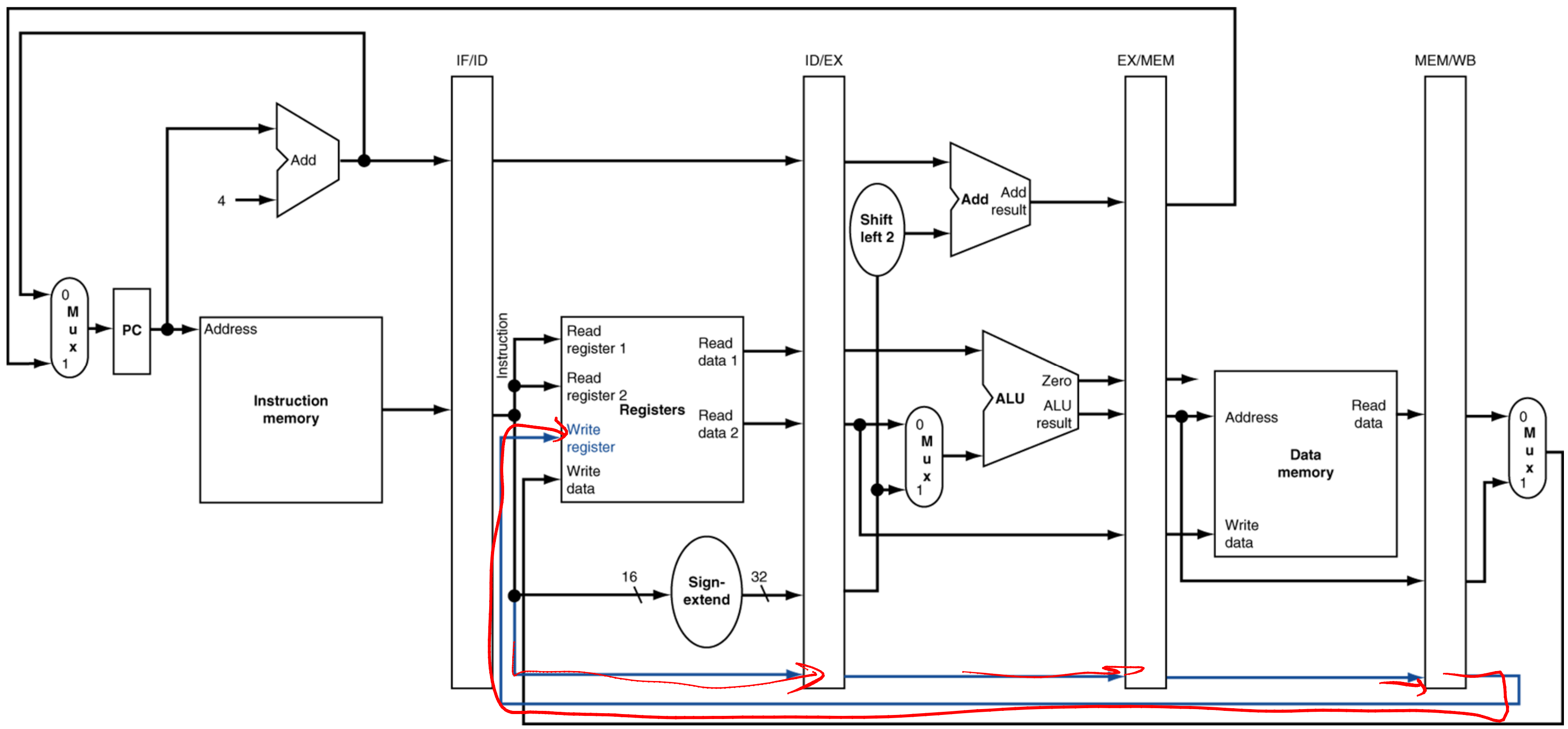
lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
0
M
u
x
1
PC
Address
Instruction
memory
Instruction
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
16
Sign-
extend
32
Shift
left 2
Add
Add
result
ALU
Zero
ALU
result
Address
Data
memory
Write
data
Read
data

WB for Load

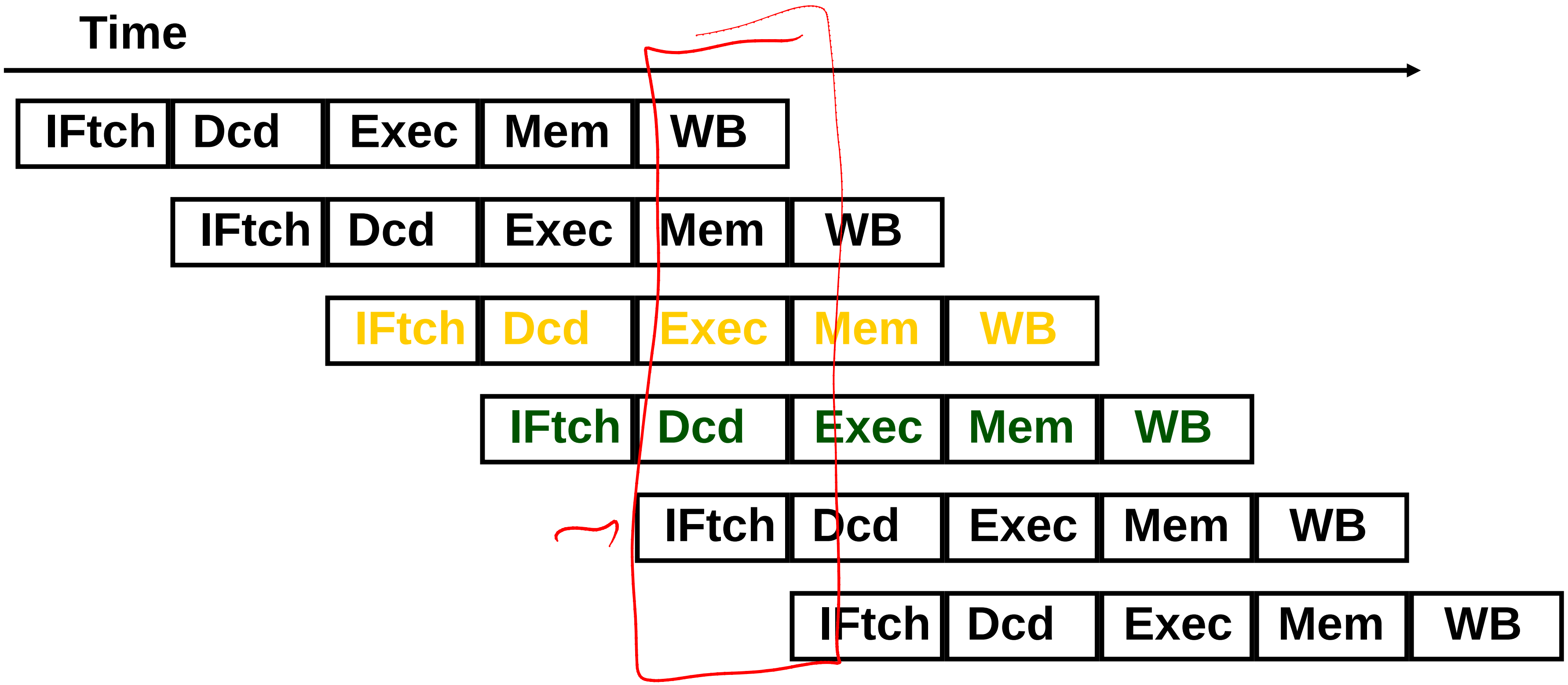

IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
IFtch Dcd Exec Mem WB
lw
Write back


IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add
Add result
Shift left 2
0 M u x 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
0 M u x 1
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
0 M u x 1
Wrong register number

Corrected Datapath for Load



Pipelined Execution Representation



- Every instruction must take same number of steps, also called pipeline "stages", so some will go idle sometimes

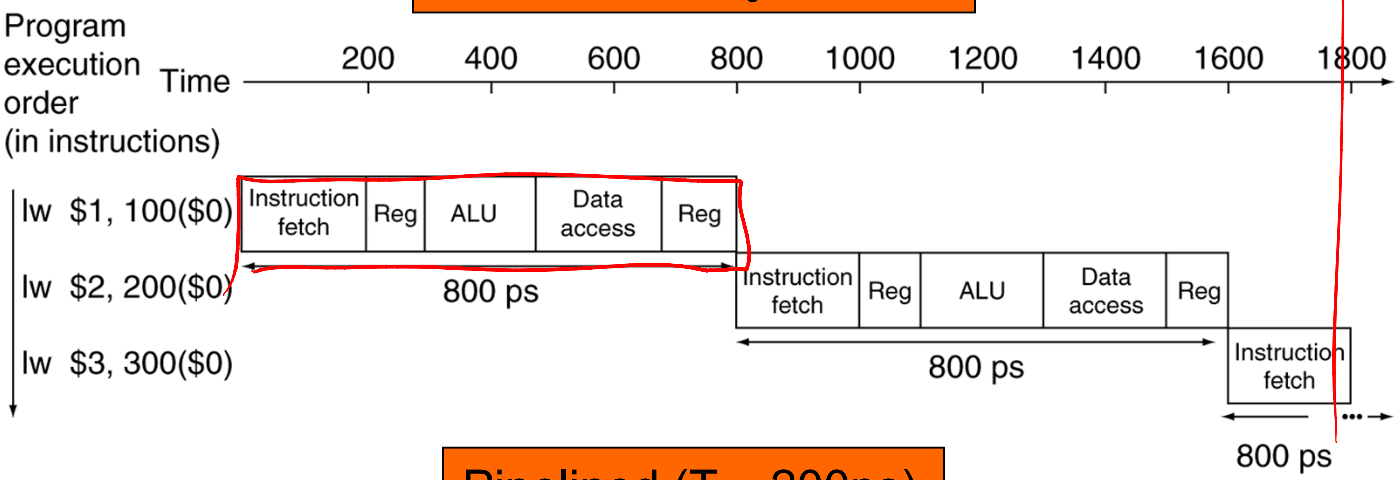
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- What is pipelined clock rate?
  - Compare pipelined datapath with single-cycle datapath

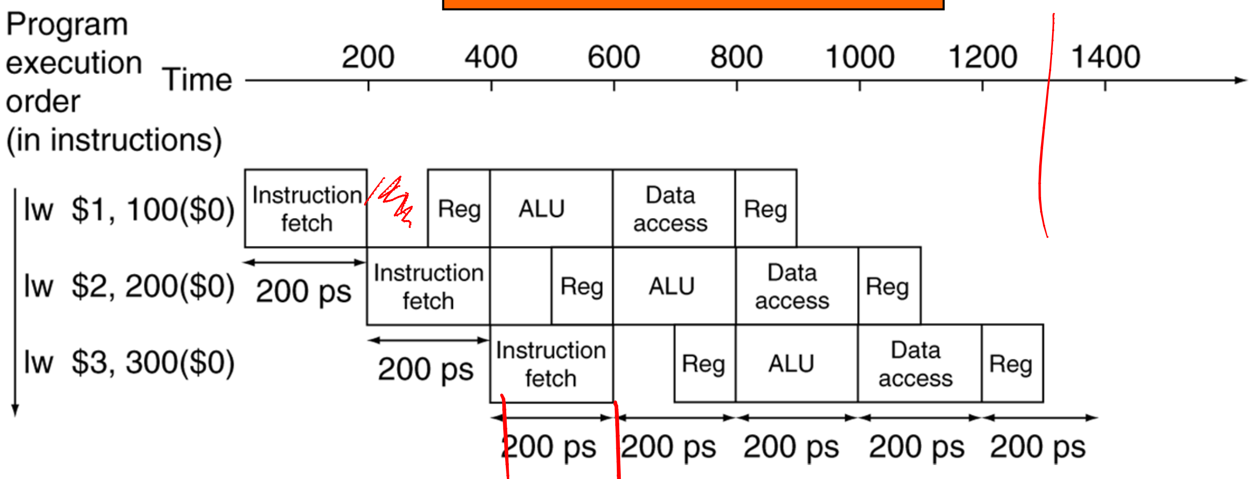
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

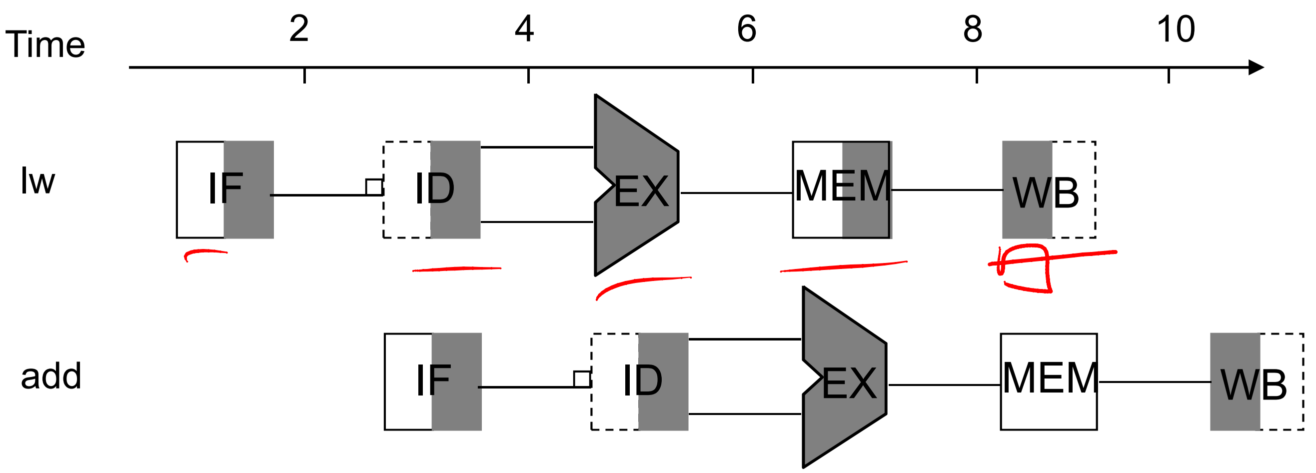
Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Graphically Representing Pipelines



- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

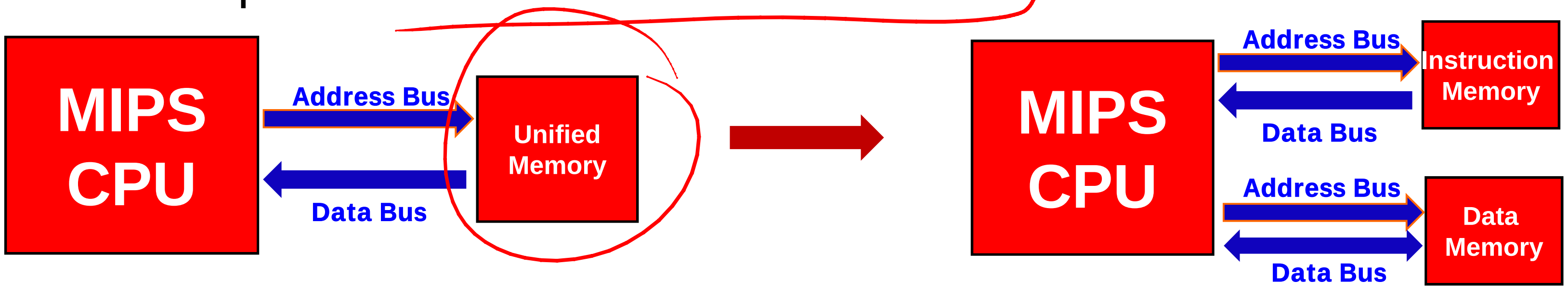
Hazards

- It would be happy if we split the datapath into stages and the CPU works just fine
  - But, things are not that simple as you may expect
  - There are **hazards**!

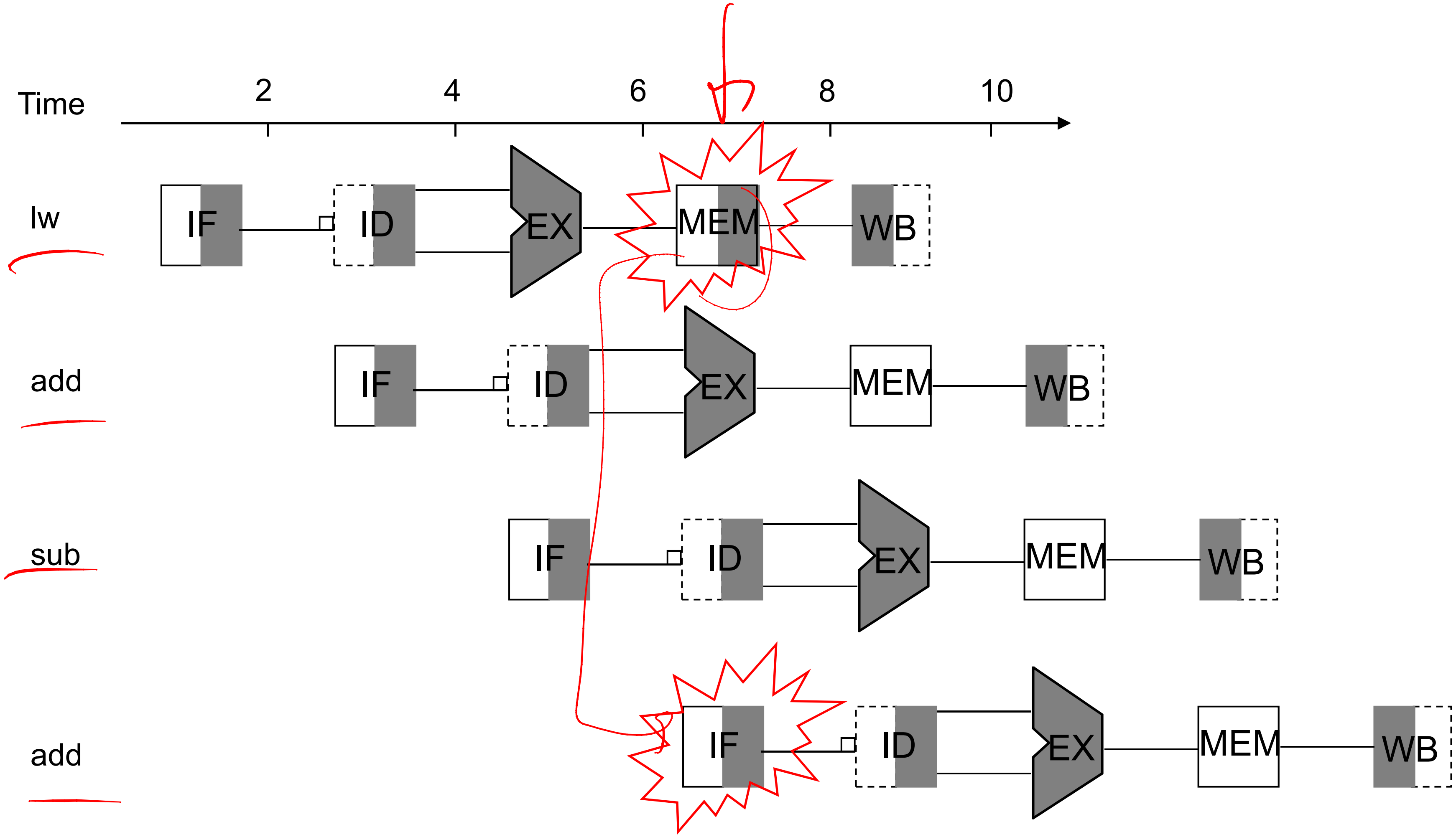
- Situations that prevent starting the next instruction in the next cycle
  - **Structure hazards**
    - Conflict over the use of a resource at the same time
  - **Data hazard**
    - Data is not ready for the subsequent dependent instruction
  - **Control hazard**
    - Fetching the next instruction depends on the previous branch outcome

Structure Hazards

- Conflict over the use of a resource at the same time
- Suppose the MIPS CPU with a single memory
  - Load/store requires data access in MEM stage
  - Instruction fetch requires instruction access from the same memory
    - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction and data memories
  - Or separate instruction and data caches



Structure Hazards (Cont.)



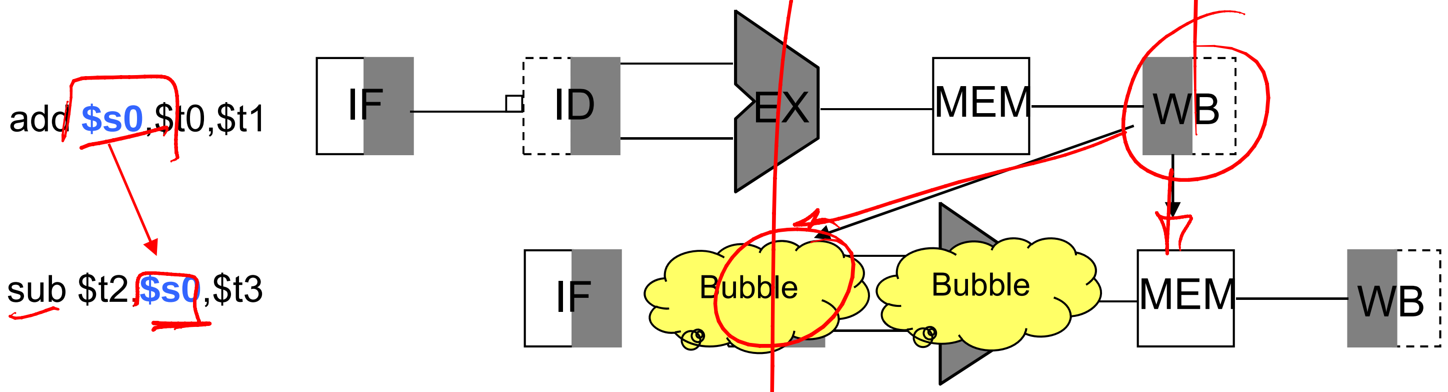
Need to separate instruction and data memory

Structural Hazard – reg read/write

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports
- Result: can perform Read and Write during same clock cycle

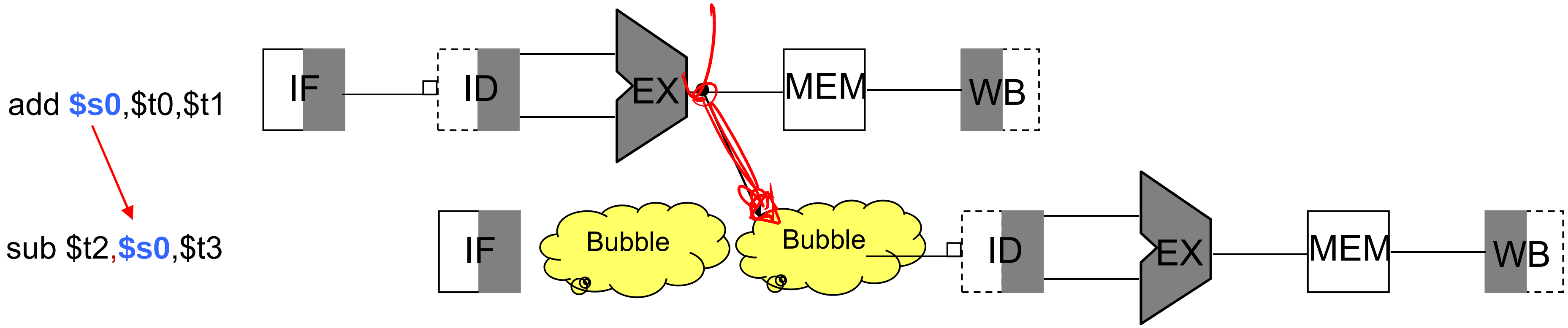
Data Hazards

- Data is not ready for the subsequent dependent instruction



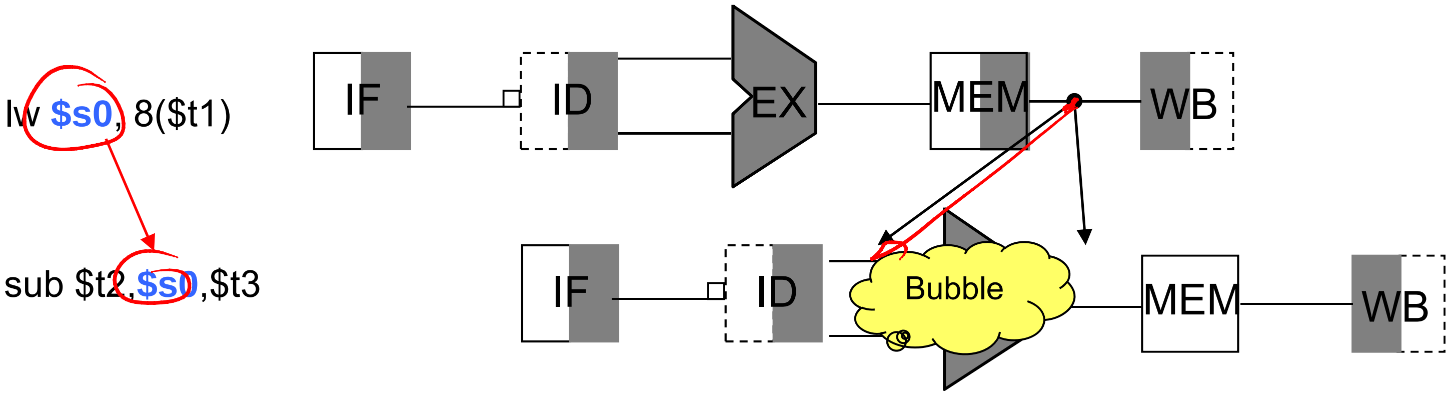
- To solve the data hazard problem, the pipeline needs to be stalled (typically referred to as “bubble”)
  - Then, performance is penalized
- A better solution?
  - **Forwarding** (or Bypassing)

Reducing Data Hazard - Forwarding



Data Hazard – Load-Use Case

- Can't always avoid stalls by forwarding
  - Can't forward backward in time!



- This bubble can be hidden by proper instruction scheduling

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A `= B + E; C = B + F;`

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2      <- stall
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4      <- stall
sw   $t5, 16($t0)
```

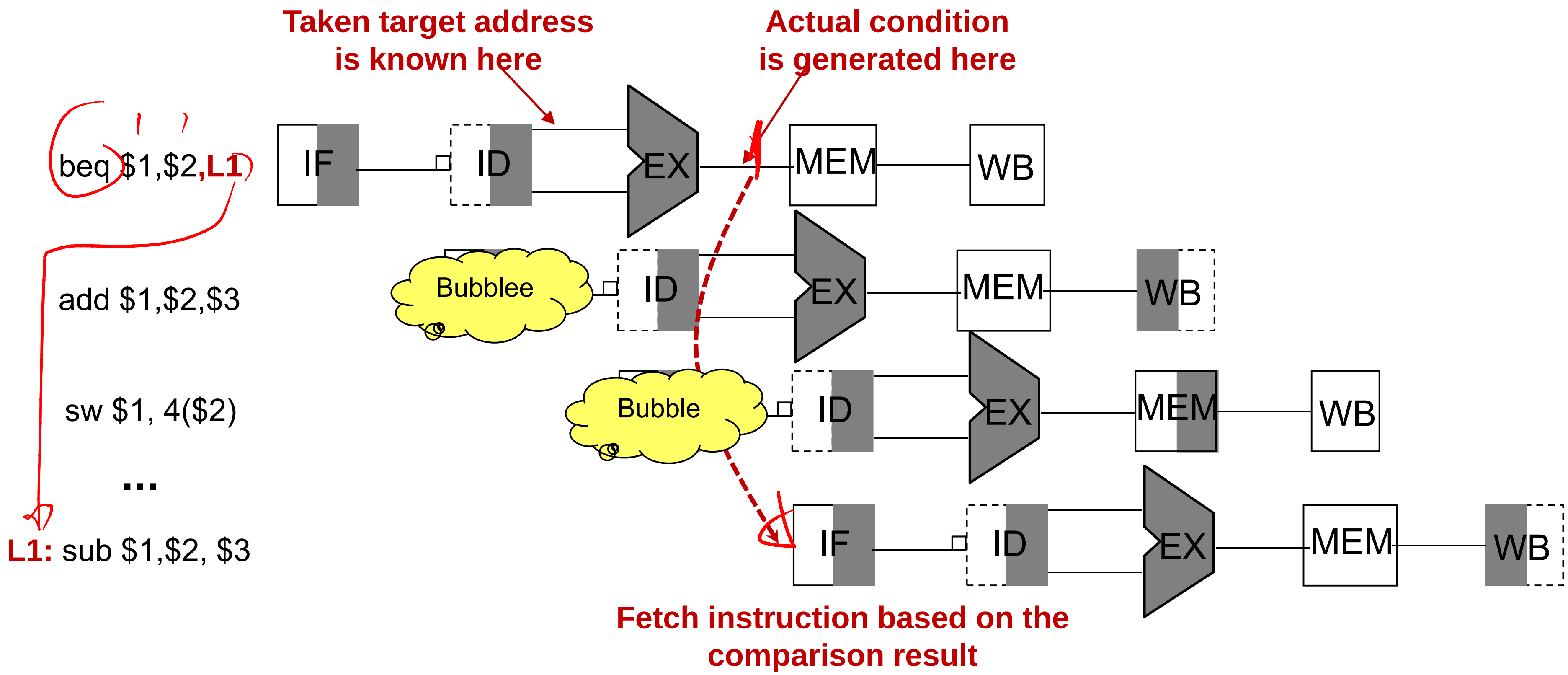
13 cycles

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

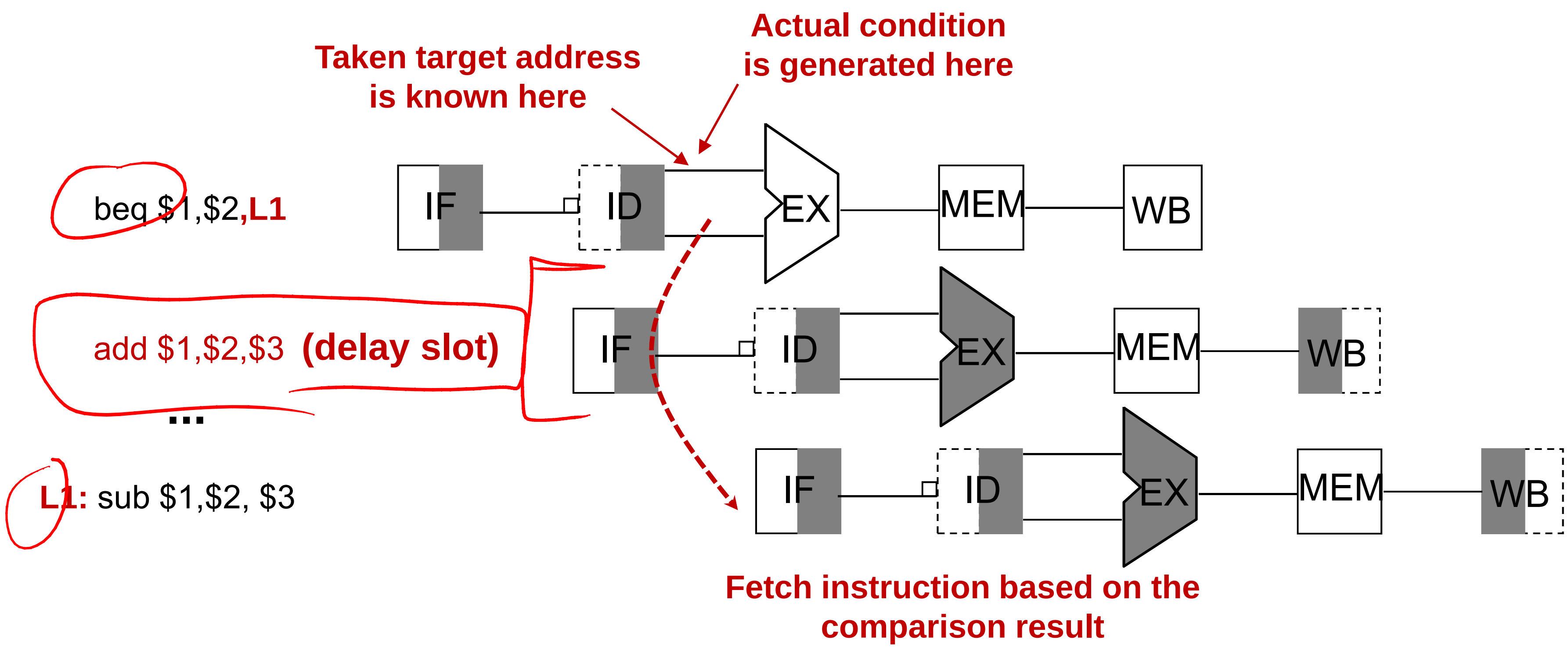
Control Hazard

- Branch determines the flow of instructions
- Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
  - Branch instruction is still working on ID stage when fetching the next instruction



Delay Slot

- Branch instructions entail a "**delay slot**"
  - **Delayed branch** always executes the next sequential instruction, with the branch taking place after that one instruction delay
  - **Delay slot** is the slot right after a delayed branch instruction



Delay Slot (Cont.)

- Compiler needs to schedule a useful instruction in the delay slot, or fills it up with nop (no operation)

```
// $s1 = a, $s2 = b, $3 = c
// $t0 = d,  $t1 = f
a = b + c;
if (d == 0)   { f = f + 1; }
f = f + 2;
```

```
      add   $s1, $s2, $s3
      bne   $t0,  $zero,  L1
      nop    // delay slot
      addi   $t1, $t1, 1
L1:  addi   $t1, $t1, 2
```

Can we do better?

```
      bne   $t0,  $zero,  L1
      add   $s1, $s2, $s3   // delay slot
      addi   $t1, $t1, 1
L1:  addi   $t1, $t1, 2
```

Fill the delay slot with a useful and valid instruction

Pipeline Summary

- Pipelining improves performance by increasing instruction **throughput**
  - Executes multiple instructions in parallel
- Pipelining is subject to hazards
  - Structure, data, control hazards
- Instruction set design affects the complexity of the pipeline implementation

Embedded Processors : examples

CISC		RISC
68000 series		Sparc
X86 family		AMD 29000
PDP-11		MIPS
VAX		SuperH
IBM 370		PowerPC
		Arm