Computer Systems Architecture

MIPS Assembler and Procedure Calls

Numbers

Arithmetic and Logic Unit

Datapaths and Microprogramming

Caches

Pipelines

Input/Output

MIPS assembler – a 'Load-Store' architecture

• Familiarity with the major aspects of the instructions set. Arithmetic instructions, branches, jumps, load and store:

`add, sub, and, or, addi, subi, li, la`

`bez, bne, beq, slt`

`j, jr, jal`

`lw, sw, lb, sb`

• R-type (3 registers), I-type (two registers) and J-type instructions. [*see diagram*]

• Addressing modes: *Immediate* (**`li, addi`**), *Register* (**`add, jr`**), *Base* (**`lw sw`**), *PC relative* (**`bne, blt`**), *Pseudodirect* (**`j`**) [*see diagram*]

• Procedure calls: MIPS register conventions: **`$t, $s, $a, $v, $sp, $ra`**

`jal, jr`

Stack use [*see diagram*]

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

©1998 Morgan Kaufmann Publishers

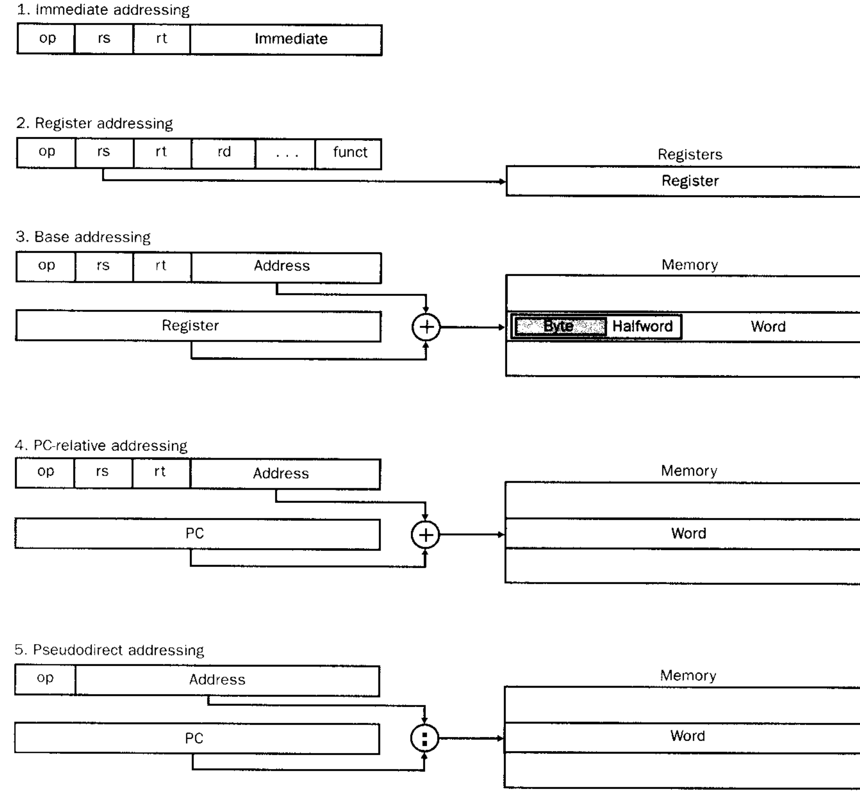


FIGURE 3.17 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1 the operand is 16 bits of the instruction itself. Modes 4 and 5 are used to address instructions in memory, with mode 4 adding a 16-bit address to the PC and mode 5 concatenating a 26-bit address with the upper bits of the PC.

Nested procedure calls

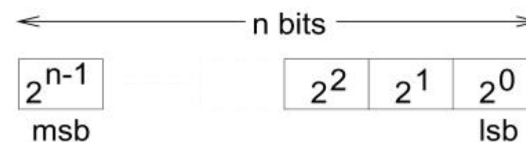
A simple example

```
A:  ...
    ...
    jal B
    ...
B:  ...
    addi $sp, $sp, -4
    sw   $ra, 0($sp)
    jal  C
    lw   $ra, 0($sp)
    addi  $sp, $sp, 4
    ...
    jr   $ra
C:  ...
    ...
    jr   $ra
```


Numbers

- **Integer representation**:

Unsigned:



Signed: *2s Compliment* (invert, +1)

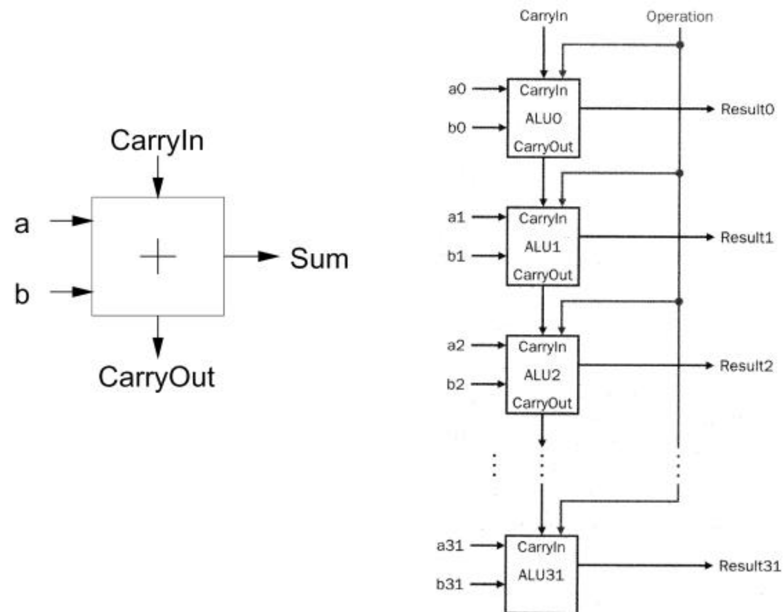
- **Floating point representation:**

value = $(-1)^s \times (1 + mantissa) \times 2^{(exponent - bias)}$

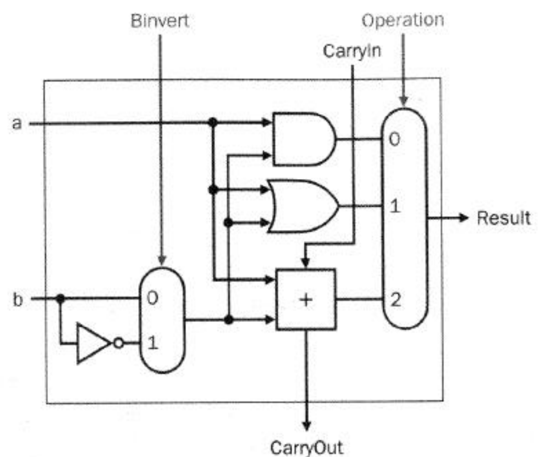
Precision errors (10 000 000 + 0.0000006 = 10 000 000*)*

Arithmetic and Logic Unit

• **Adder (**half, full, ripple, carry-lookahead)



• **ALU** (multiplexor, control lines, addition and subtraction)



• **Multiplication**

Datapaths and Microprogramming

• CPU fetches an instruction and then uses the instruction to act on (it)

• **Five stages**: 1) *Instruction Fetch* (IF) 2) *Register Read* (RR) 3) *Execute/ALU* (EXE) 4) *Memory Access* (MA) 5) *Register Write* (RW)

• **Single cycle CPU**: all logic in *one* clock cycle
[*see diagram*]

• **Multi cycle CPU**: each stage in *one* clock cycle (need intervening registers, multi-cycle control unit/FSM)
[*see diagram*]

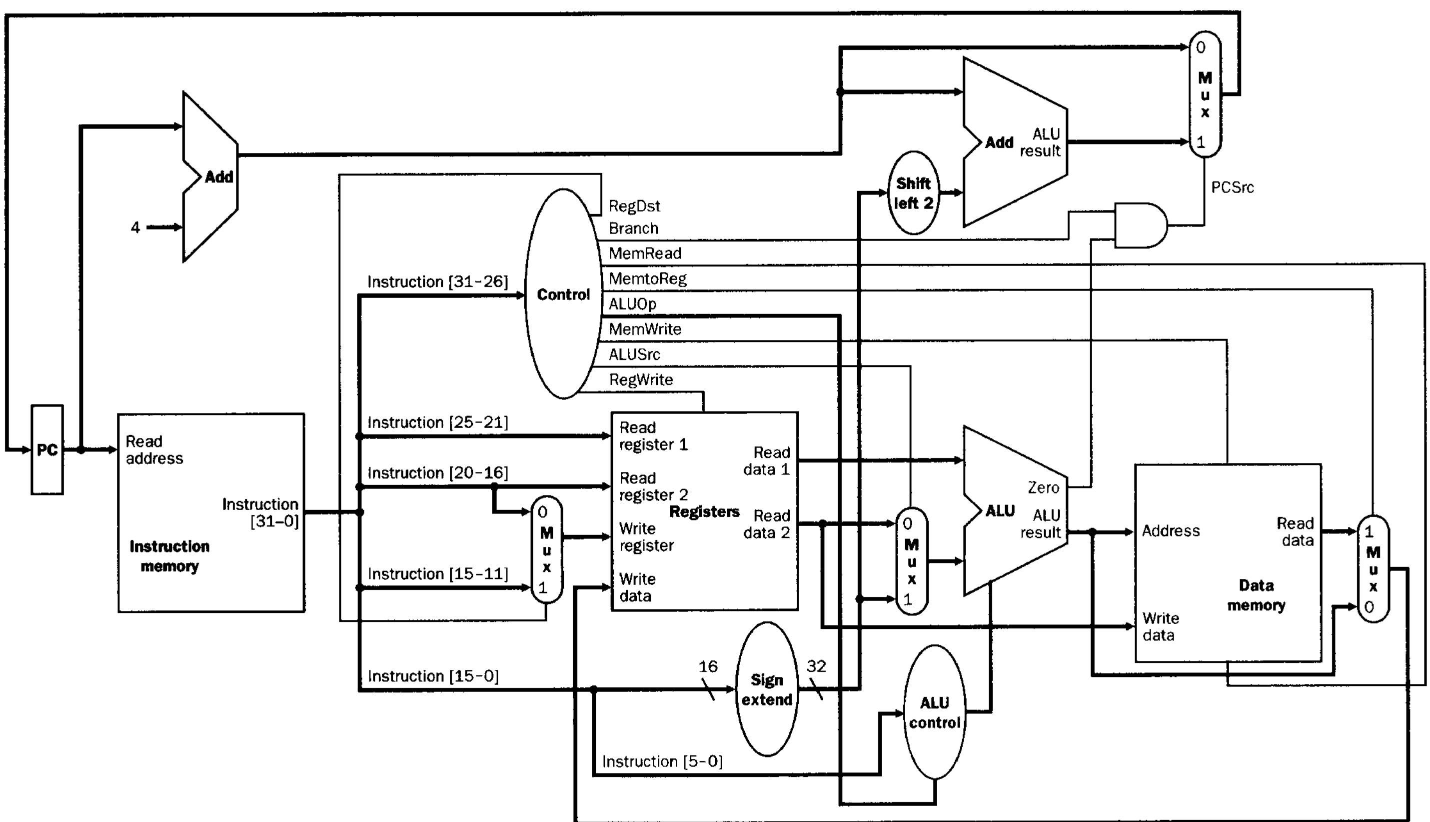


FIGURE 5.19 The simple datapath with the control unit. The input to the control unit is the 6-bit opcode field from the instruction. The outputs of the control unit consist of three 1-bit signals that are used to control multiplexors (RegDst, ALUSrc, and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus we drop the signal name in subsequent figures.

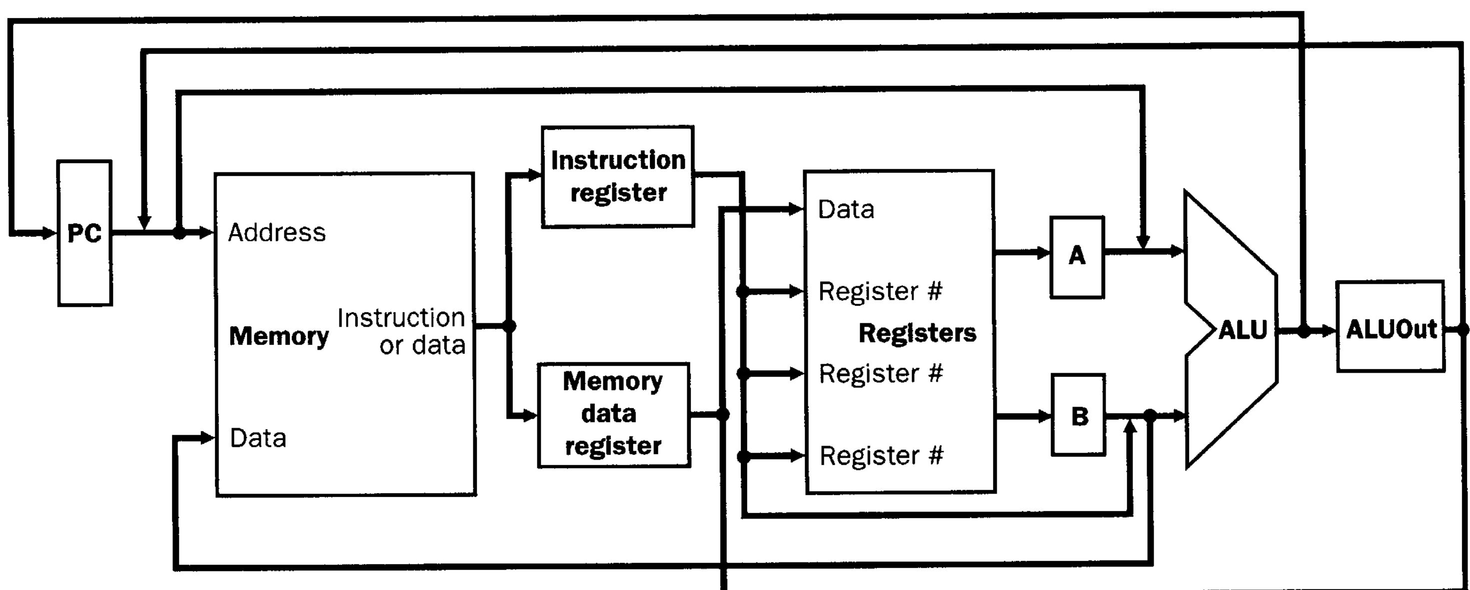


FIGURE 5.30 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

Caches

Memory hierarchy: why? [*see diagram*]

Direct mapped Cache: *modulo mapping*, tags, valid bits, block size

Fully Associative Cache: logic too complicated

Set-Associative Cache: Compromise

Write Strategies: write-through, write-back

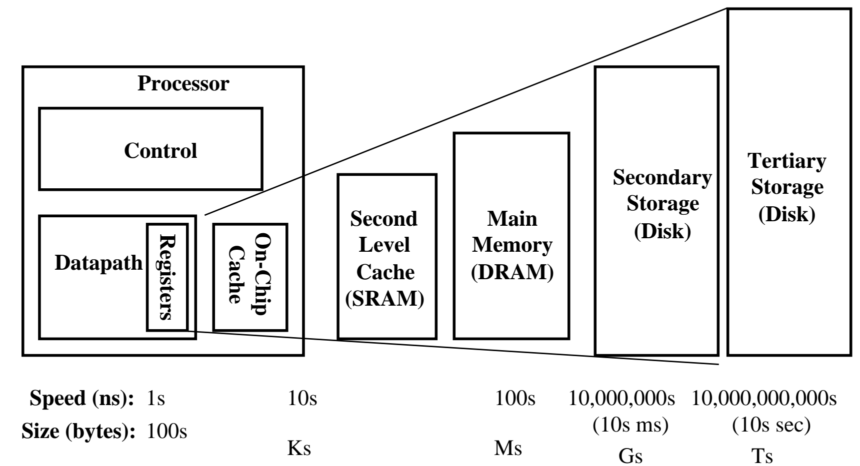
Replacement Strategies: Least recently used (needs h/w), random

Cache Misses: *Cold start*, *Capacity* (full up), *Conflict* (set full up)

Hazards: Data hazards (`lw, add`); Control hazards (`bne`)

Memory Hierarchy of a Modern Computer System

- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.



DAP Fa97, © U.CB

Pipelining

Pipelining: Each stage (e.g. 5) of the CPU runs one instruction

Instruction Fetch	**IF**
Register Read	**RR**
Execute (ALU)	**EXE**
Memory Access	**MEM**
Register Write	**RW**

[*See diagram*]

Hazzards: *Structural*: Hardware can't support instruction combination

Control: Branch instructions

Data: Need previous (adjacent) calculations/fetches

Branch Prediction: Avoid control hazards (sometimes)
– Predict, 'always fail'
– Use 'branch table' like cache to predict

Forwarding: Use clever logic in datapath to get at results *before* instruction completion

[*See diagram*]

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
Store word (sw)	2 ns	1 ns	2 ns	2 ns		7 ns
R-format (add, sub, and, or, slt)	2 ns	1 ns	2 ns		1 ns	6 ns
Branch (beq)	2 ns	1 ns	2 ns			5 ns

FIGURE 6.2 Total time for eight instructions calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

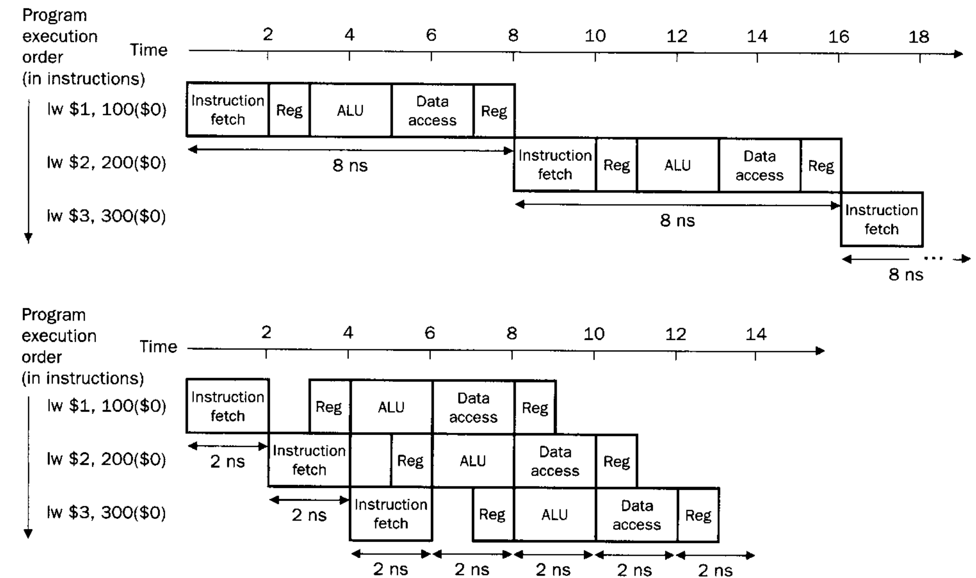


FIGURE 6.3 Single-cycle, nonpipelined execution in top vs. pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 6.2. In this case we see a fourfold speedup on average time between instructions, from 8 ns down to 2 ns. Compare this figure to Figure 6.1. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The computer pipeline stage times are limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

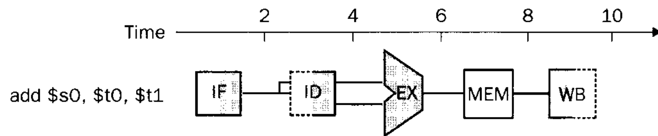


FIGURE 6.7 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 6.1 on page 437. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write back stage, with the drawing showing the register file being written. The shading indicates the element is used by the instruction. Hence MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

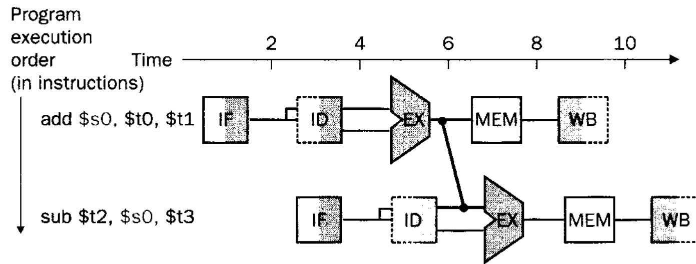


FIGURE 6.8 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

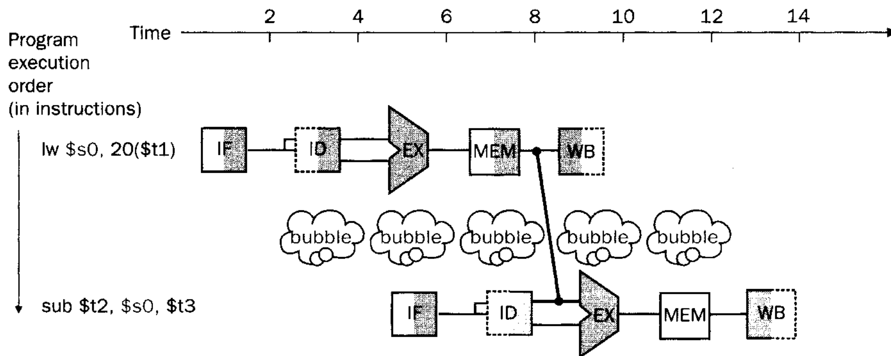
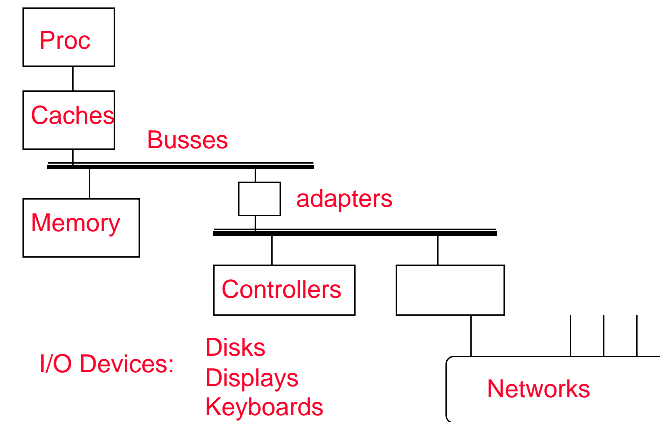


FIGURE 6.9 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backwards in time, which is impossible.

Summary: Computer System Components



° **All have interfaces & organizations**