Chapter 7: MIPS Processor Design

Topics:

Overview of components
Simple MIPS implementation
Performance

Basic concepts of Pipelining

Reading: Patterson and Hennessy
4.1 – 4.5 (skim 4.6)

Basic concepts

Datapath: `passive part of circuit`
`storage elements (registers, RAM)`
`glue logic`

Control: `active part; finite state machine`
`with control signals`
`control the datapath components`

Goal: design a simple CPU to support a subset of MIPS instructions

MIPS subset:

R-format	add, sub and or slt
load/store	lw, sw
branches	beq (j)

Simple single-cycle implementation

Each instruction completes in one (long!) cycle.

(Cycle must be long enough for slowest operation to complete.)

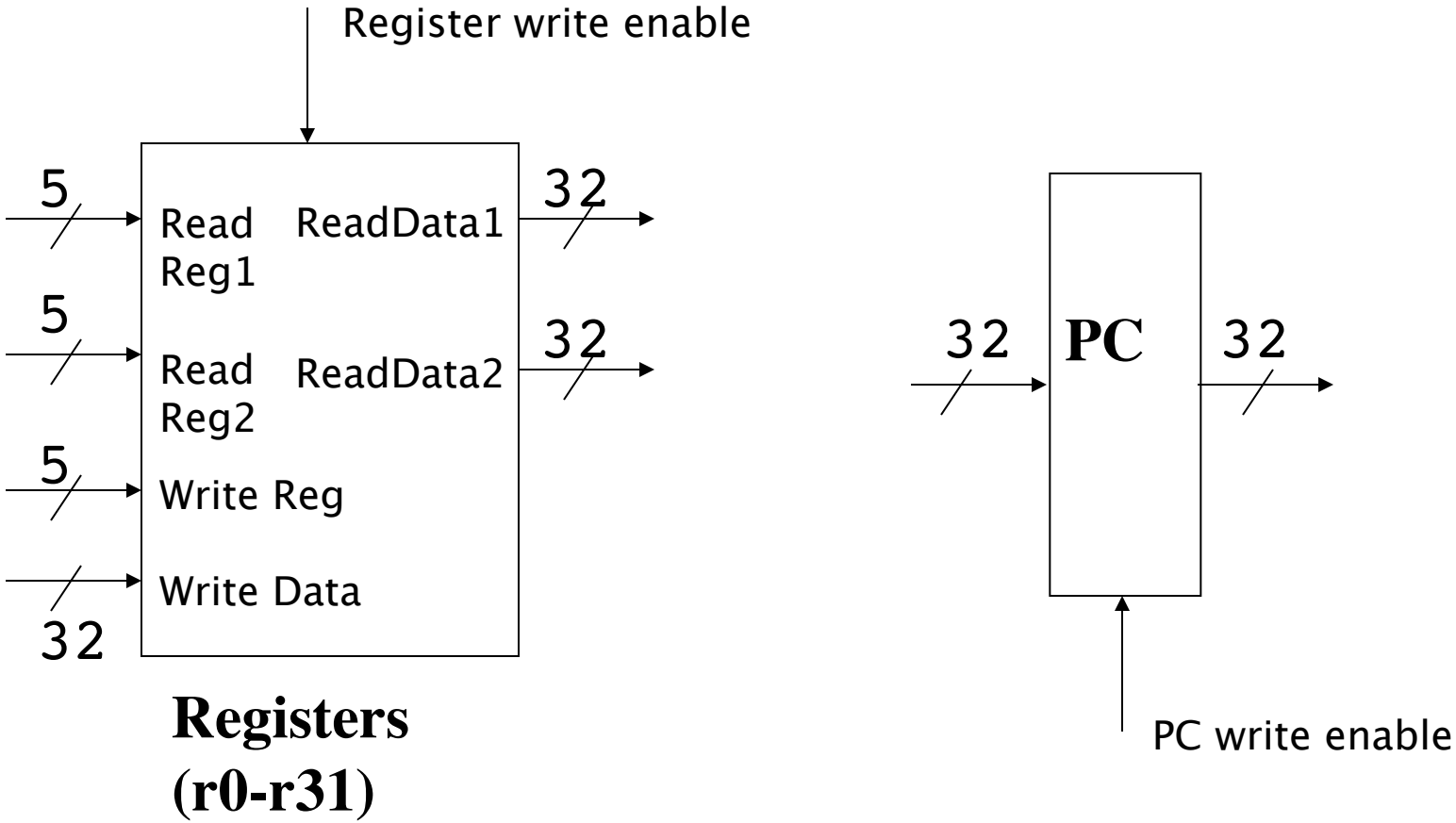
Advantage: `fairly simple design`

Disadvantage: `fast instructions have to wait for slow instructions to finish`

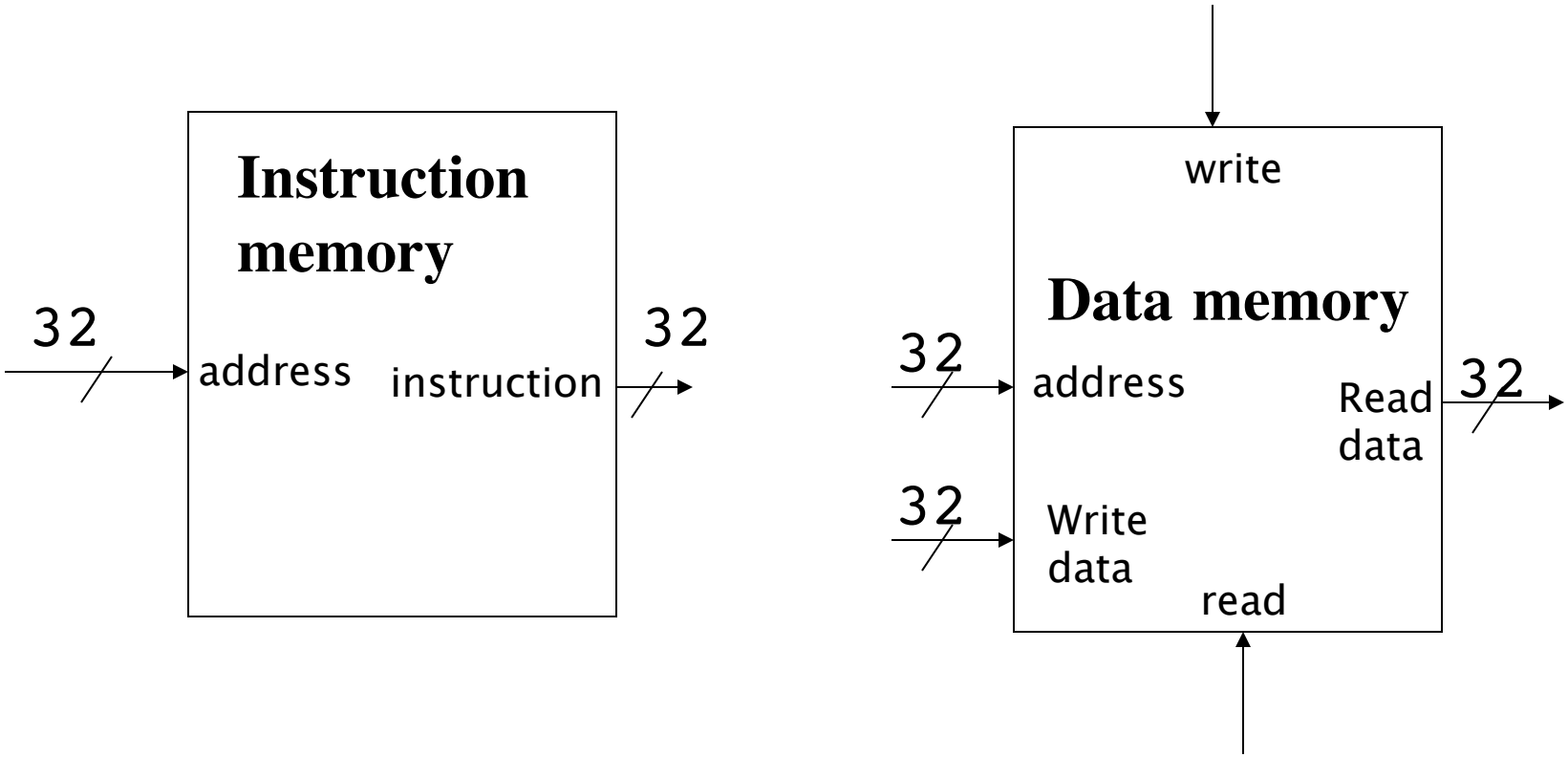
Review: MIPS instruction formats

p. 136 Fig. 2.20

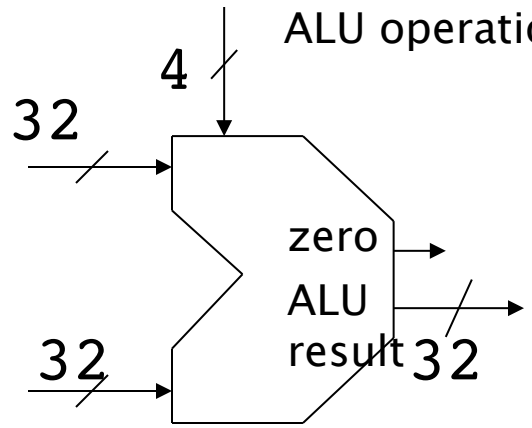
Major components: registers



Major components: memory



Major components: arithmetic/logic unit



ALU operation	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	Set less than
1100	NOR

Zero:

```
0 if ALUResult !=0
1 if ALUResult == 0
```

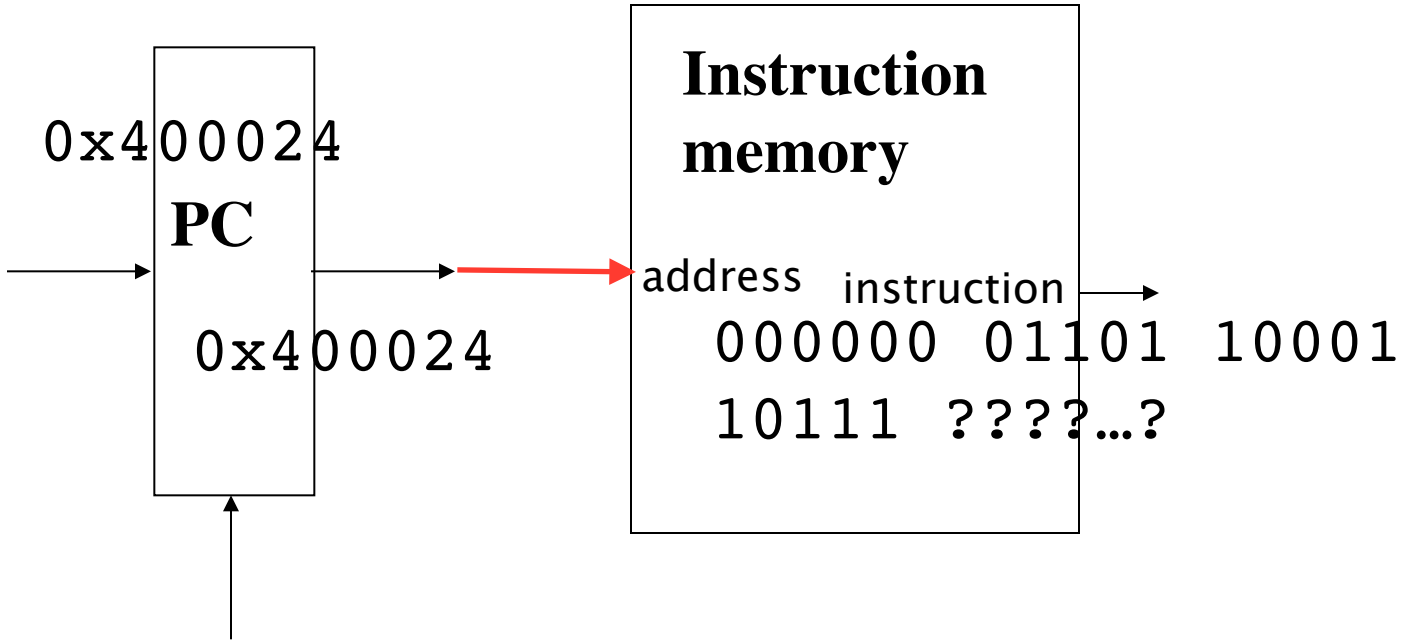
Processing an arithmetic/logic instruction (R-format; all register operands)

0x400024

Ex.: add $23, $13, $17

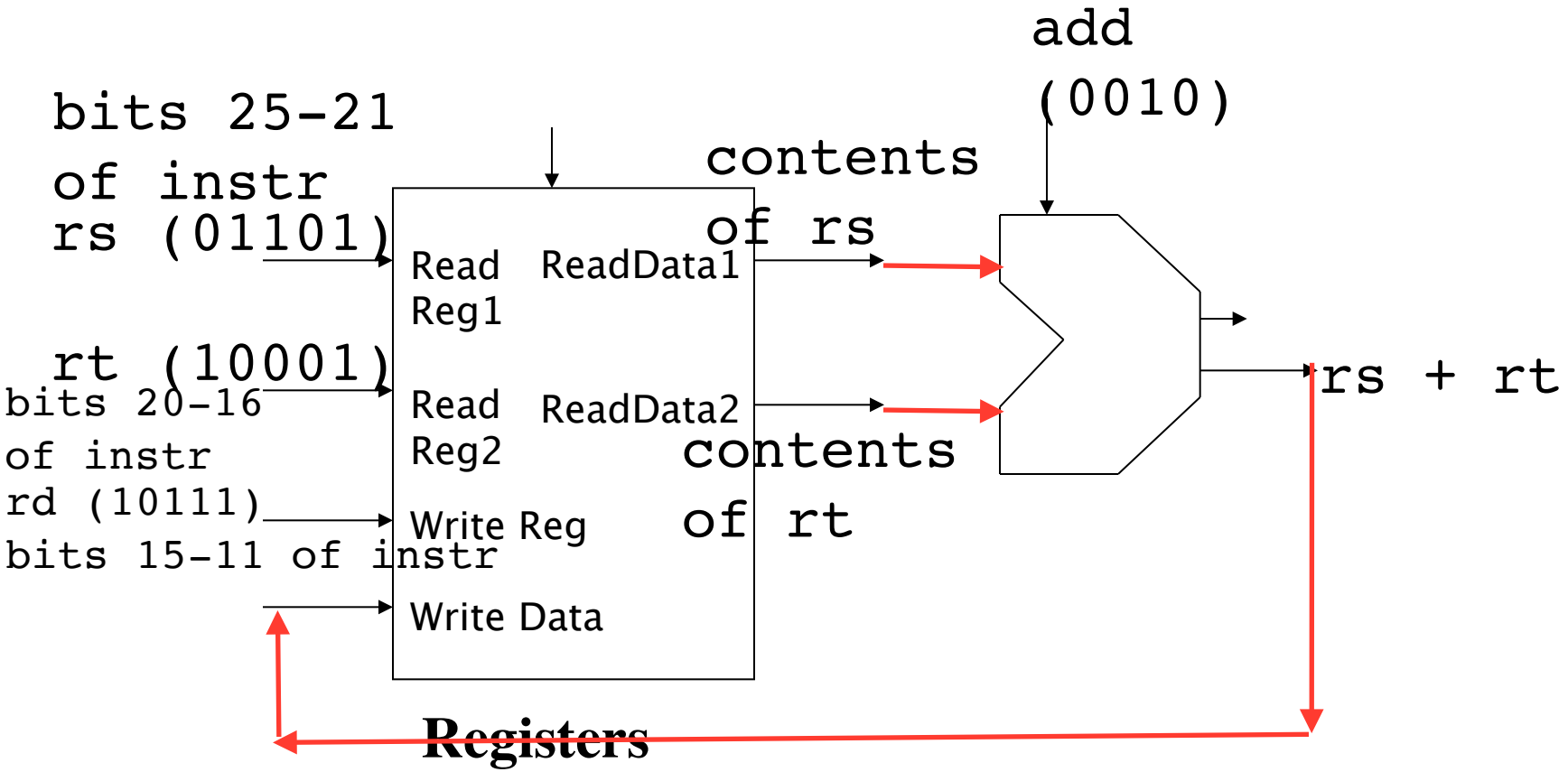
rd rs rt

Step 1: fetch instruction from memory

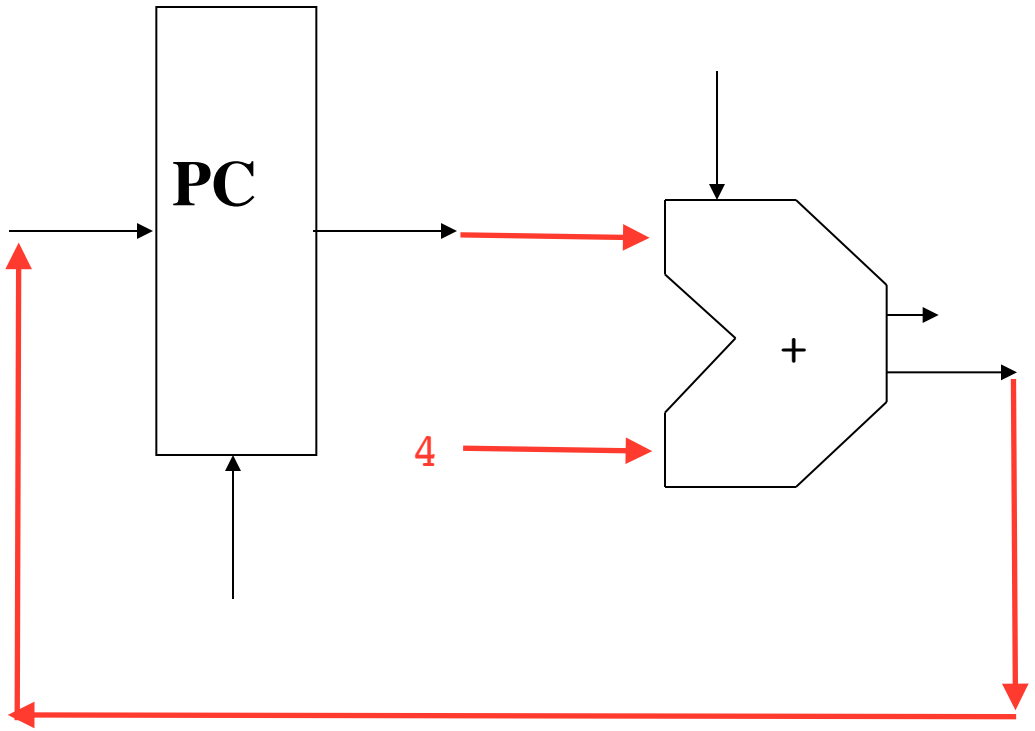


Step 2:

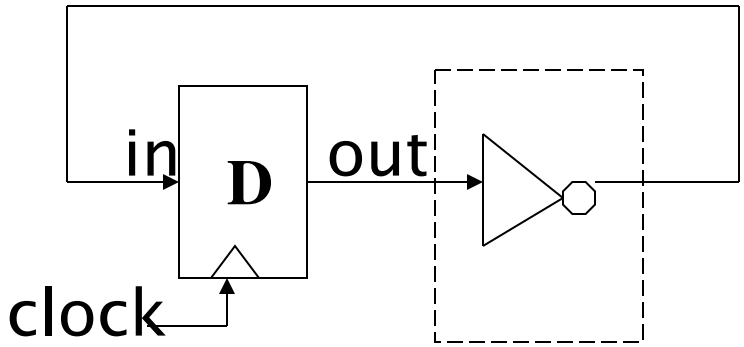
```
read rs, rt
execute add operation
write result in rd
```



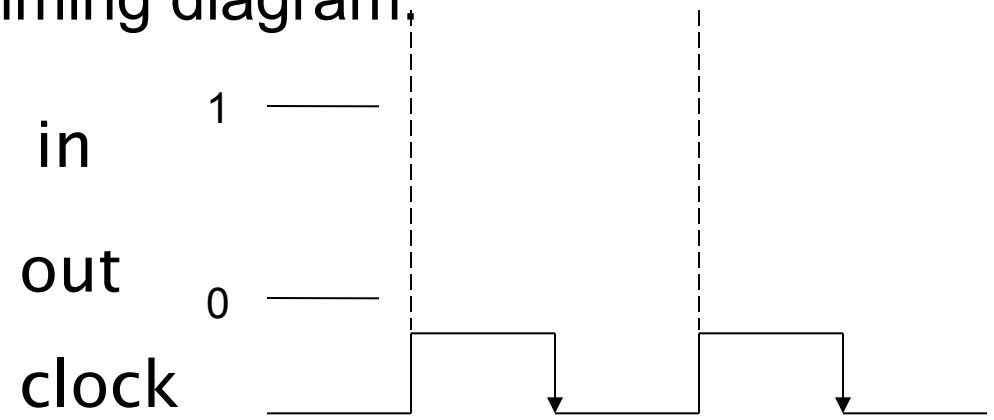
To fetch next instruction:



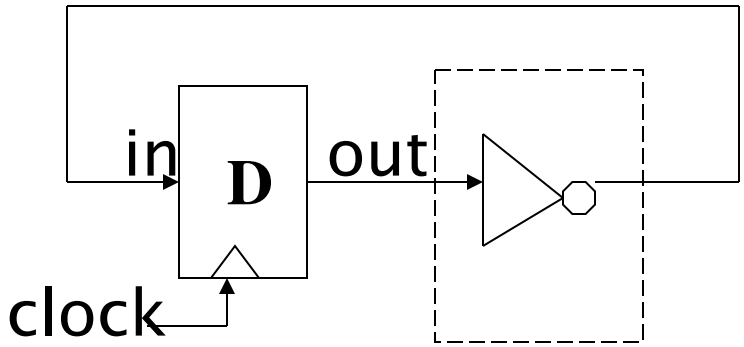
Basic operation of edge-triggered logic:



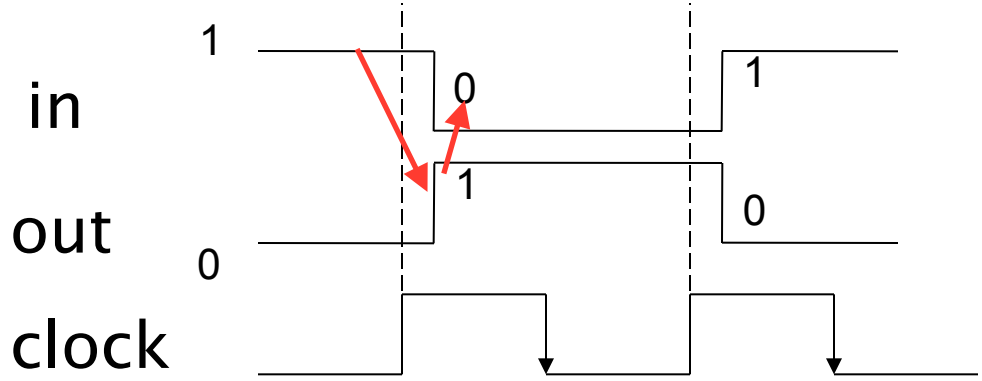
Timing diagram:



Basic operation of edge-triggered logic:



Timing diagram:



Processing a lw instruction

Ex.: lw $23, -4($13)
rt rs

Step 1: `fetch instruction from memory`

Step 2: `read rs, rt`

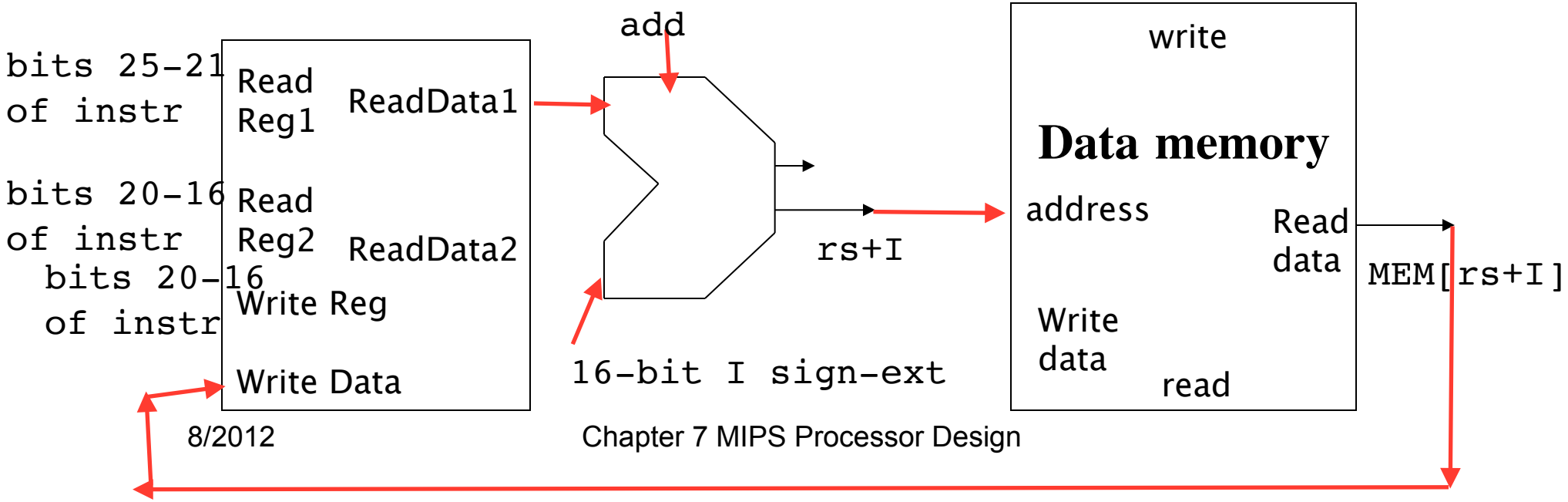
`Text`

Processing a lw instruction (con't)

Step 3: `ADDR = rs + 16-bit constant sign-extended`

Step 4: `read MEM[ADDR]`

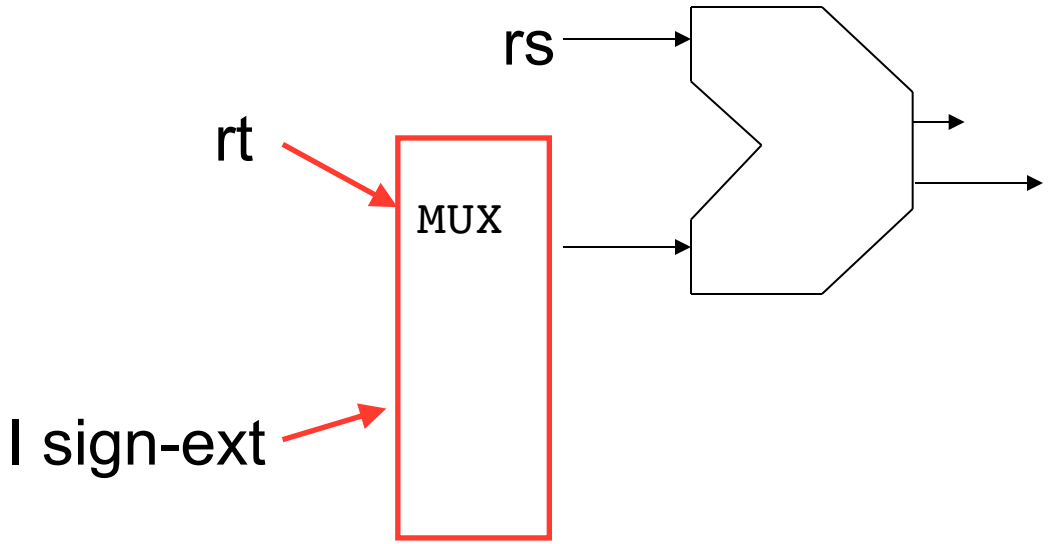
Step 5: `rt = MEM[ADDR]`



R-format instructions: rt goes into lower ALU input

load/store instructions: 16-bit constant goes into ALU input

Solution:



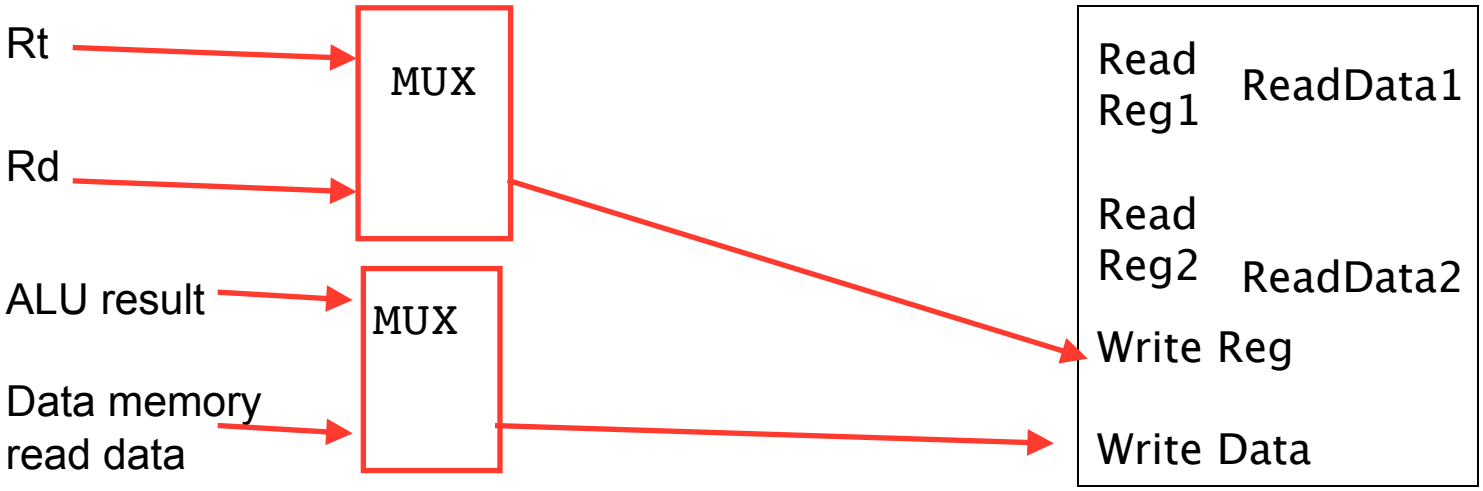
R-format instructions:

ALU result to *write data* of registers
rd to *write register* of registers

load instructions:

data memory *read data* to *write data* of registers
rt to *write register* of registers

Solution:



Beq (branch if equal) `beq rs, rt, I`

`if (rs == rt) PC = BTA; else PC = PC + 4`

Two operations:

1) `BTA = PC+4 + 16-bit I shift left 2, sign-extended`

2) `if (rs - rt == 0) PC = BTA`

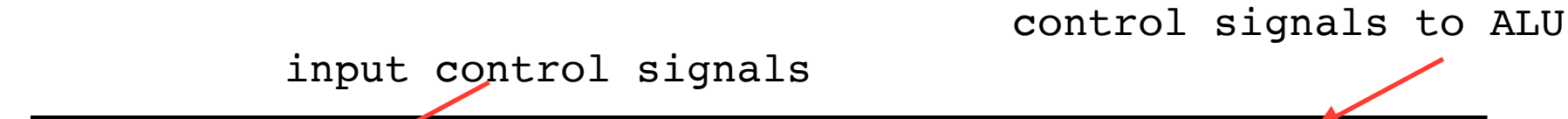
Simple datapath to support MIPS subset:

p. 320 Fig. 4.15

List of control signals for setting datapath options:

```
RegDst: 0 write to rt
        1 write to rd

RegWrite: 0: don't write
          1: write to register

ALUSrc: 0 contents of rt to ALU lower input
        1 16-bit I sign-ext to ALU lower input

MemRead: 0 don't read mem; 1 read mem

MemWrite: 0 don't write mem; 1 write mem
MemtoReg: 0 ALUResult written to register
          1 data from memory written to register
Branch:  0 not BEQ instruction
        1 BEQ instruction

ALUOp (2 bits):
00 ALU performs add always
01 ALU performs subtract always
10 ALU follow operation for R-format instructions
```

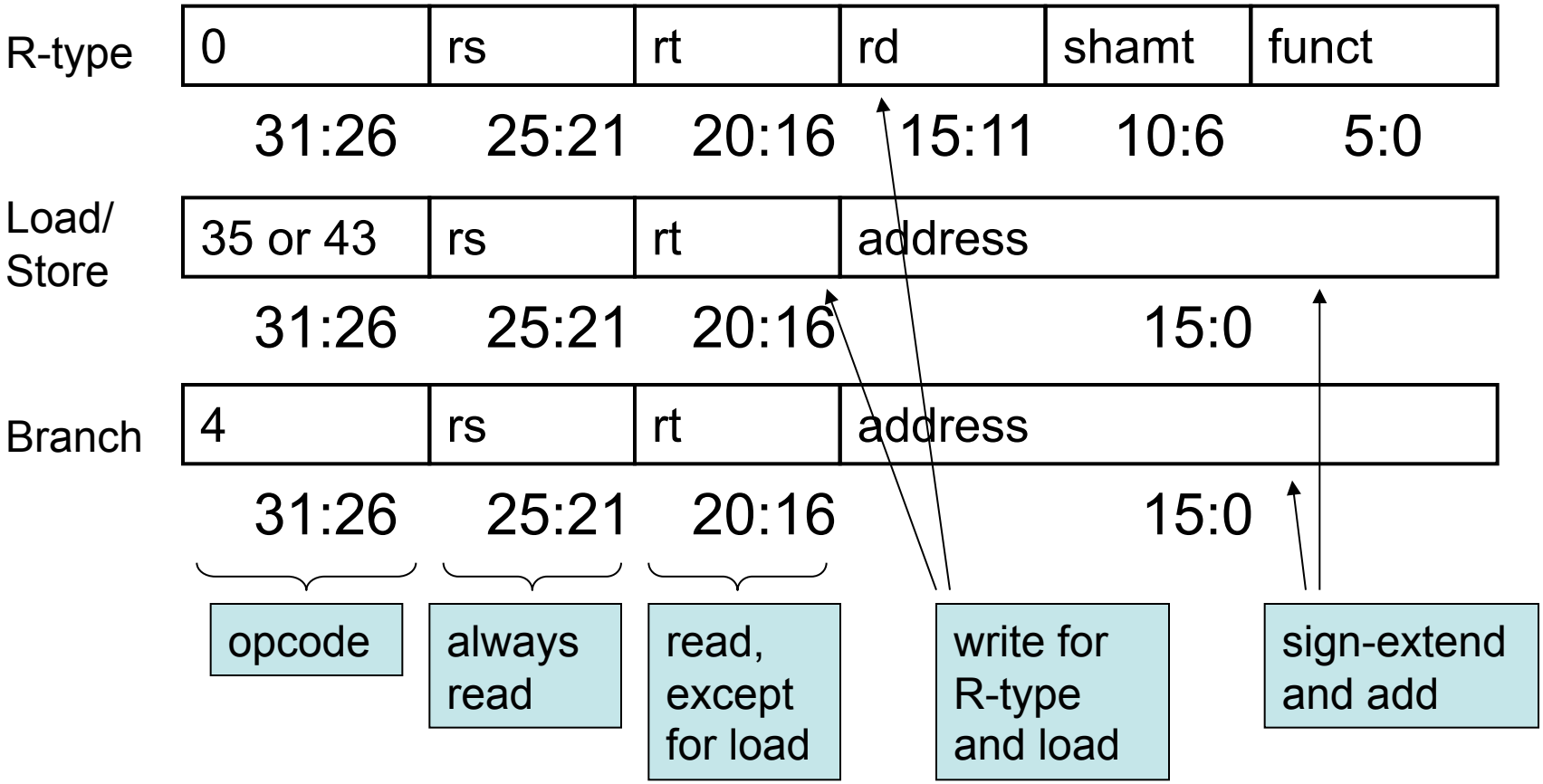
What ALU has to do depends on instruction:



opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Different parts of instruction word determine control signals. 2-bit ALUOp determined by opcode.

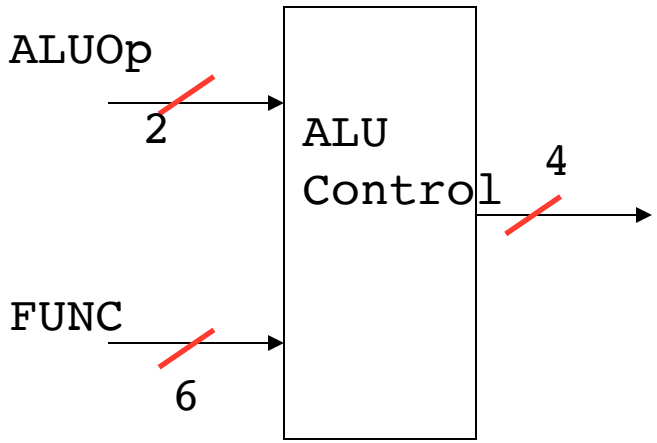
For R-type, must also look at instr[5-0].



ALU control signals in more detail

ALUOp definition:

ALUOp	ALU action
00	Force ALU to add
01	Force ALU to subtract
10	Follow instr[5-0]



See p.317 Fig. 4.12
p. 318 Fig. 4.13

Determining control signals for each instr

instr	Reg Dst	ALU Src	Mem To Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Op0
R-format	1	0	0	1	x	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	x	0	1	0	1

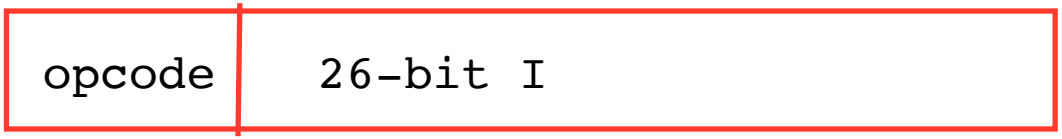
R-type instruction (p. 324 Fig. 4.19)

Lw instruction (p. 325 Fig. 4.20)

Beq instruction (p. 326 Fig. 4.21)

Adding the jump instruction

Jump instruction format:



Necessary operations:

1) `target address = top 4 bits of PC || 26-bit I || 00`

2) `PC = target address`

Jump instruction (p. 329 Fig. 4.24)

Performance of single-cycle datapath

Given latencies:

Memory units 200 ps, ALU/adder 200 ps, register read 100 ps, register write 100 ps

R-format: `200ps(instr fetch)+100ps(reg read)+200ps(ALU) + 100ps (reg write) = 600ps`

Lw: `200ps + 100ps + 200ps + 200ps + 100ps = 800ps`

Sw: `200ps + 100ps + 200ps + 200ps = 700ps`

Beq: `200ps + 100ps + 200ps = 500ps`

J:

`Single cycle implementation means one cycle = 800 ps`

Cycle time versus clock rate

Suppose your home computer has a 1 GHz clock.

1 GHz = `10^9 cycles per second`
1 cycle = `10^-9 seconds`

Since 1 ns = `10^-9 seconds`
and 1 ps = `10^-12 seconds`

For our single-cycle implementation,
clock rate = `1.25 GHz`

Pipelining: basic concepts

Doing laundry in 4 steps (each 30 minutes):

- Wash
- Dry
- Fold
- Put in closet

Each load takes 2 hours. 4 loads take 2 * 4 hours.

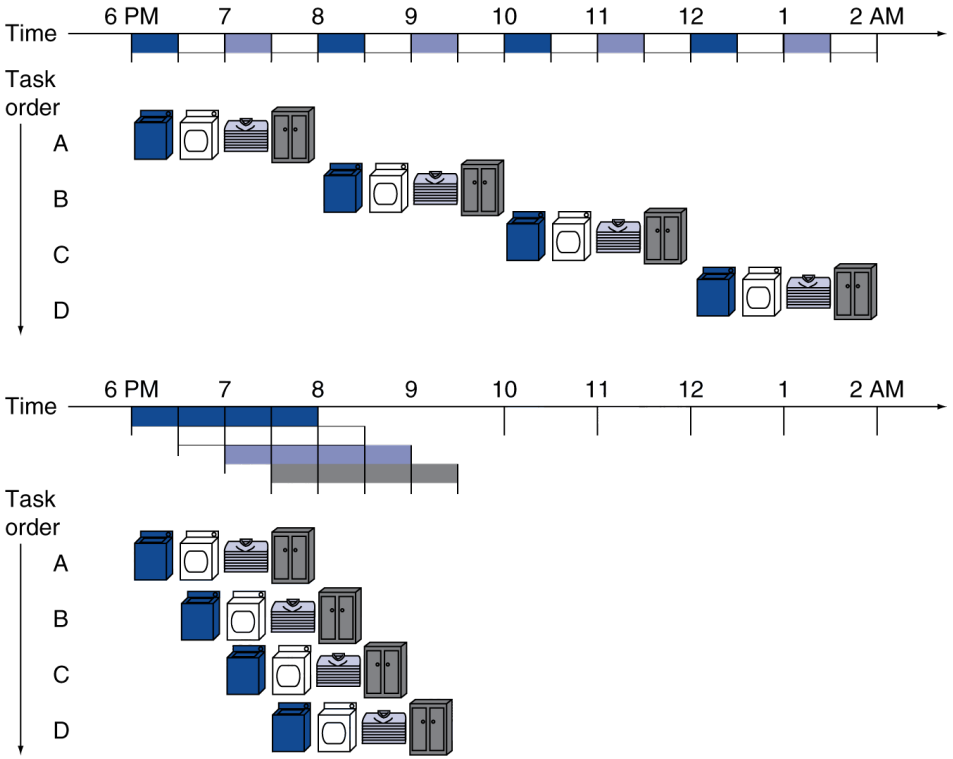
But can *overlap* steps, like an assembly line.

- Wash load 1
- Dry load 1, wash load 2
- Fold load 1, dry load 2, wash load 3, etc

4 loads take `3.5` hours

Speedup = `longer time / shorter time = 8/3.5`

If N loads (very large N), speedup is close to 4 (number of steps)!



Simple MIPS pipeline

Consider lw instruction (slowest). 5 steps:

- Instruction fetch (IF): get instruction from memory
- Instruction decode/register fetch (ID): generate control signals, get rs, rt
- Execute (EX): calculate memory address
- Memory access (MEM): read data from memory
- Writeback (WB): write result back to rt

(Somewhere: PC = PC + 4)

Each step uses a different part of the datapath!

- IF uses instruction memory
- ID reads registers, generates control signals
- EX uses ALU
- MEM uses data memory
- WB writes registers

Can overlap steps: *pipelined* MIPS implementation.

`stopped 11/10/2016`

What's the speedup? Remember:

- Register read or write take 100 ps
- Memory access, ALU take 200 ps

Without pipelining:

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

With pipelining, the duration of each step must be equal to the duration of the *slowest* step.

Hence, the cycle time for pipelining is (p. 334 Fig. 4.27):

Cycle 1: lw $1… in IF
Cycle 2: lw $2… in IF
lw $1… in ID
Cycle 3: lw $3… in IF
lw $2… in ID
lw $1… in EX
Etc, etc

Cycle 5: lw $1… in WB (other lw's in IF, ID etc)
Cycle 6: lw $2… in WB (other lw's in IF, ID etc)

If there are many instructions, *ideally* each instruction takes 1 cycle!

MIPS ISA makes it easy to implement a pipeline!

Each MIPS instruction is 32 bits.

- In IF, get 32 bits (MEM[PC]) from memory, one instruction.

  (later: x86 instructions have different sizes)
- To get next instruction, PC = PC + 4
- Instruction format is fixed; easy to decode

Load/store architecture:

- Lw/sw access data memory once per instruction

Pipeline hazards

Ideally, start/complete new instruction every cycle in pipeline.

But sometimes this may not be possible: *hazards*

Hazards may result in *stalls* or *bubbles* in pipeline (cycles where nothing happens in a stage)

3 types of pipeline hazards:

- Data hazards
- Control hazards
- Structural hazards

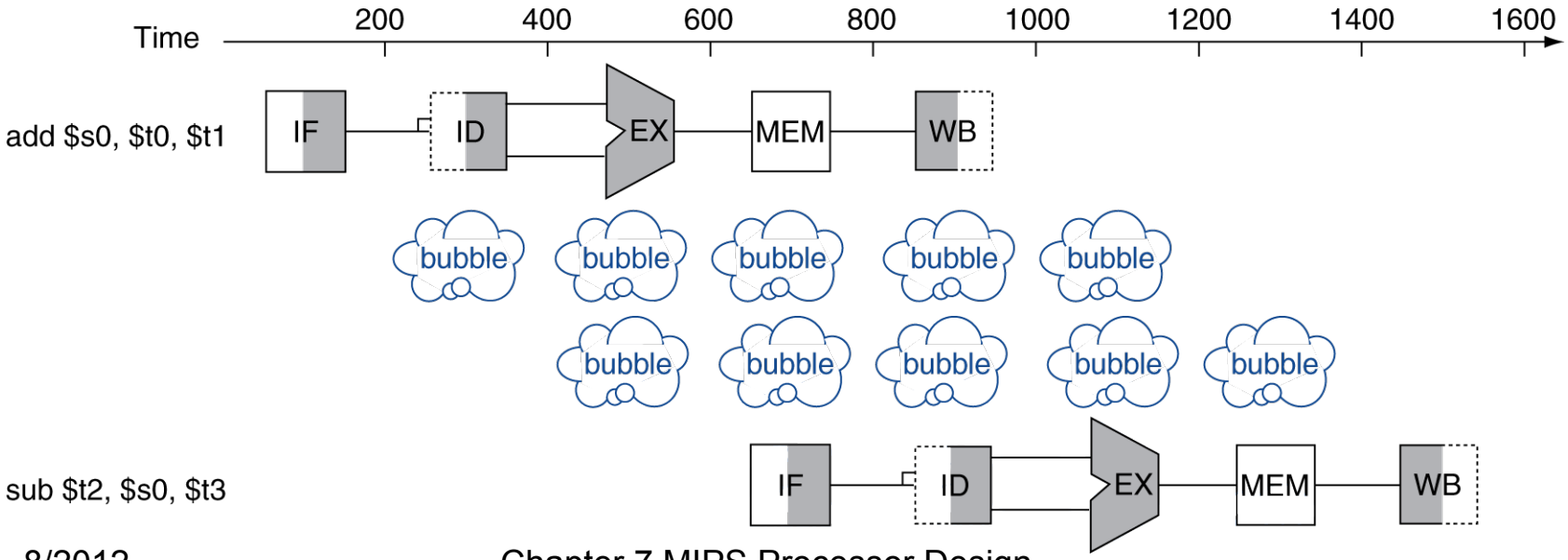
Data hazards

An instruction uses the result from a previous instruction. Ex.:

add $s0, $t0, $t1

sub $t2, $s0, $t3

$s0 is only ready in Cycle 5! Sub has to wait:



However, result of add is computed at the end of Cycle 3.

- Can *forward* result to sub, *bypassing* $s0; no stall!
- Needs extra logic and connections in hardware

p. 337 Fig. 4.29:

Load-use data hazards

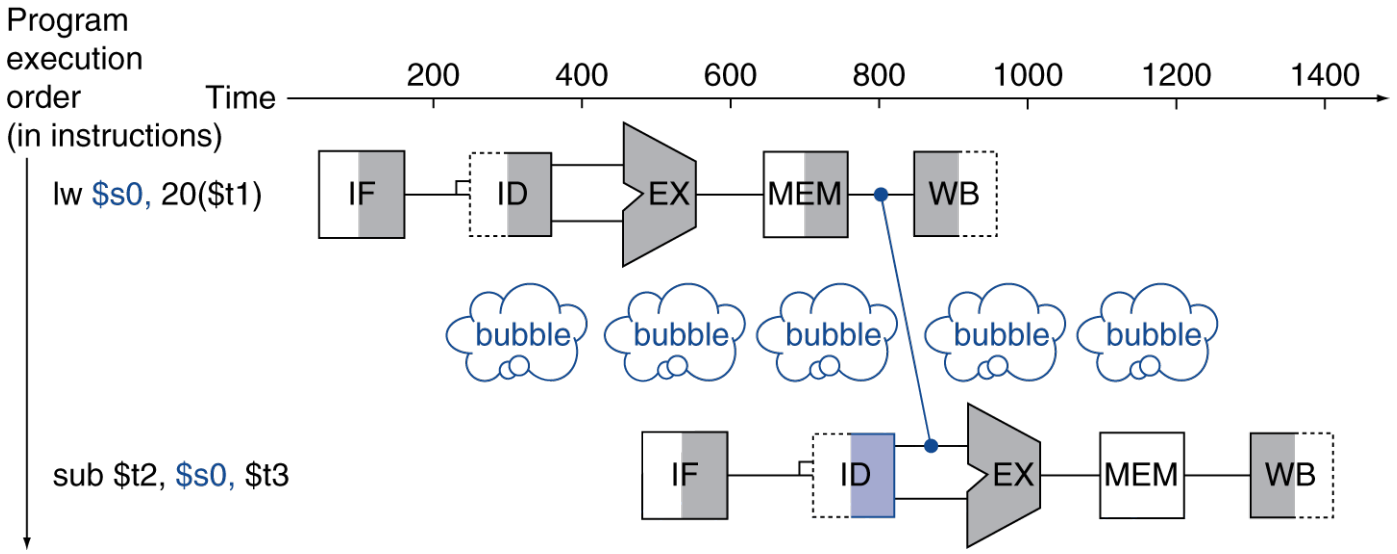
Load-use example:

	1	2	3	4	5
lw $s0, ?	IF	ID	EX	MEM	WB
add ?, $s0, ?					

Result of lw ready at the end of Cycle 4.

Add needs result of lw at the beginning of Cycle 3.

This results in a *load-use* data hazard. A stall/bubble is *necessary*.



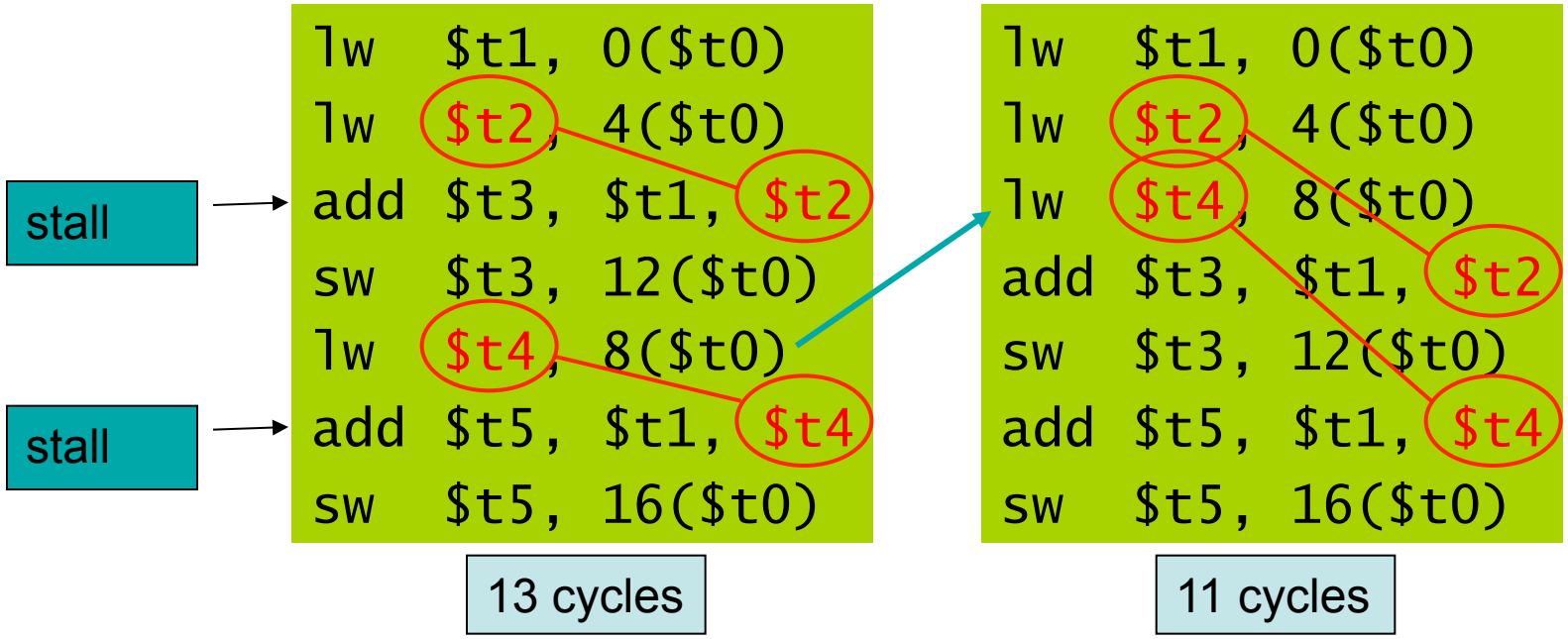
More likely timing:

cycle	1	2	3	4	5	6	7
lw $s0…	IF	ID	EX	MEM	WB		
sub $t2…		IF	ID	stall	EX	MEM	WB

At cycle 4, sub is blocked in ID, lw continues to MEM.

Code scheduling to avoid load-use stalls

Sometimes, a compiler can *re-order* or *reschedule* an instruction sequence to avoid load-use stalls.



Control hazards

Control hazards are pipeline hazards caused by instructions that change the PC (conditional branches and jumps). Ex.:

beq $s0, $s1, T

A: *[fall-through]*

T: *[target]*

Cycle 1: beq in IF

Cycle 2: what instruction to fetch?

Need to know as soon as possible:

- Branch target address for beq
- $s0 == $s1?

Can try to calculate both in ID stage.

But stalls are often necessary.

Can also try to *predict* branches.

In general, branches/control instructions are difficult for efficient pipeline operation.

Structural hazards

A structural hazard occurs when a hardware component is needed by more than one instruction, in the same cycle.

This is a *resource conflict*, or *resource contention.*

Current MIPS pipeline has no structural hazards.

But suppose we decide to combine instruction and data memories into one memory.

Then on every lw/sw, when we access data in the MEM stage, there is a conflict with the IF stage.

This would be a *structural hazard;* the pipeline has to stall.

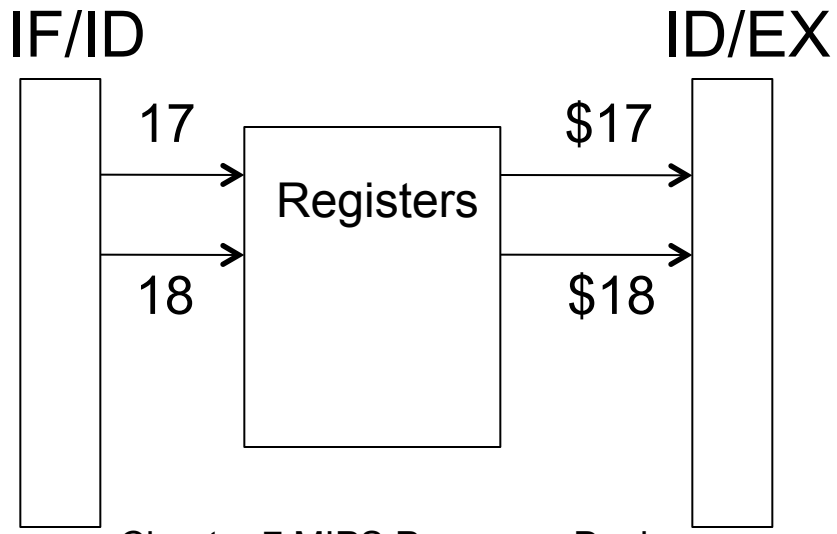
Simple MIPS datapath, divided into stages

p. 345 Fig. 4.33 (needs one more small change):

Need pipeline registers between each pair of stages.

- At rising edge of clock, new contents appear in register before each stage
- New contents go through logic in stage, computations are performed
- New contents propagate to input of next register

Ex.: add $16, $17, $18



Full view (p. 355 Fig. 4.41):

Ex.: lw $16, 0($24), IF

Ex.: lw $16, 0($24), ID

Ex.: lw $16, 0($24), EX

Ex.: lw $16, 0($24), MEM

Ex.: lw $16, 0($24), WB

Note: each instruction carries its control signals (and other necessary information) with it through the pipeline!

Control signals, register numbers, memory addresses etc are pulled out of pipeline registers and used in the appropriate stages.

All instructions have the same operations in IF and ID:

Stage	**operations**
IF	Current instr = MEM[PC]; PC = PC + 4
ID	Read rs, rt; generate control signals

Each instruction type has own operations after ID:

Stage	lw	sw	ALU	beq
EX	ADDR = rs + I	ADDR = rs + I	Result = rs op rt	Compute BTA; rs - rt
MEM	Data = MEM [ADDR]	MEM[ADDR] = rt		If (rs-rt == 0) PC = BTA
WB	rt = Data		rd = result	

A more complex trace:

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

Trace: Cycle 1

Trace: Cycle 2

Trace: Cycle 3

Trace: Cycle 4

Trace: Cycle 5

Simplified view of pipeline control (p. 359 Fig. 4.46):

Carrying information between stages (p. 361 Fig. 4.50):

Performance

Measuring performance (and cost) is not trivial.

For example, we are in downtown SF, and need to get to the airport. 2 options:

- Take BART
  - About $8 per person
  - Takes about 30 minutes
  - Small carbon footprint
- Take a cab
  - About $30 per cab
  - Takes 15-20 minutes (more if bad traffic)
  - Large carbon footprint

Similarly, when comparing performance of two computer systems, we need to know the operation environment, and what we are measuring.

Suppose we are interested in running an application to perform a task. Ex., convert CD tracks to MP3.

The application can convert multiple tracks at the same time.

Definitions:

- *Response time*: how long it takes from the beginning to end of a task

  [measure time from beginning to end of a conversion]
- *Throughput*: how many tasks completed per unit time

  [measure how many tracks converted per hour]

Relative performance and speedup

For now, we'll focus on response time.

Common situation: two systems X and Y run the same application. Which is faster?

We measure the response time on the two systems.

Suppose X has a shorter response time.

The speedup (always > 1) is defined as

S = larger execution time / smaller execution time

= Execution time on Y / Execution time on X

We say:

X is S times faster than Y.

Example: X takes 10 seconds, Y takes 12 seconds

X is 12/10 = 1.2 times faster than Y

More terms:

Elapsed time: total time for executing task, including I/O, operating system time, idle time

CPU time: time spent by the CPU only on task

Unix *time* command gives 3 times:

- Real (elapsed)
- System (CPU time spent in OS)
- User (CPU time spent in user code)

Estimating performance of a program

Suppose we are interested in the performance of an application (say, MP3 encoder).

Run the app, with performance monitoring tools. Can count:

instruction count (IC): total no. of instructions executed

percentage of each type of instruction, for example:

- 20% loads
- 10% stores
- 55% arithmetic/logic
- 15% branches/jumps

We'll run the MP3 encoder on our *base PC*.

Given: IC = 10^9

Given: estimates for each instruction type:

- each load takes 1.5 cycles
- each store takes 1.2 cycles
- each ALU instruction takes 1 cycle
- each branch takes 2 cycles

Total execution time in cycles =

- # loads * 1.5 +
- # stores * 1.2 +
- # ALU * 1 +
- # branches * 2

=

Suppose our PC has a 2 GHz clock.
Each cycle is: ps

Total execution time on base PC =

How to get better performance?
Suppose we buy a better compiler for our base PC.
We recompile the MP3 encoder using the new compiler.

The new compiler is very effective at reducing the number of load-use and control stalls.

(No other changes.)

Each load now takes 1.2 cycles (instead of 1.5).

Each branch/jump now takes 1.5 cycles (instead of 1.8).

What is the new execution time?

time spent on loads =

time spent on stores =

time spent on ALU =

time spent on branch/jumps

total time =

Speedup =

CPI (Cycles per instruction)

Earlier example:

loads take 1.5 cycles

stores take 1.2 cycles, etc

These are the *CPI* (cycles per instruction) measurements for each instruction type.

We can also talk about average CPI for a program

average CPI = no. of cycles / IC

CPI Example

Suppose we compile a program. There are 3 classes of instructions (A, B, C). Which is faster?

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

No. of cycles for sequence 1 =

Avg CPI for sequence 1 =

No. of cycles for sequence 2 =

Avg CPI for sequence 2 =

It’s important to look at execution time!
IC or CPI by itself does not give enough information.
In general:

Execution time = IC * avg. CPI * cycle time
    where IC = no. of instructions executed
        avg. CPI = no. of cycles executed / IC
        cycle time = length of each cycle in seconds

Performance depends on:

- Algorithm
  affects IC, CPI
- Programming language
  affects IC, CPI
- Compiler
  affects IC, CPI
- ISA
  affects IC, CPI, cycle time
- Hardware implementation
  affects CPI, cycle time

Power consumption

CMOS integrated circuits (ICs) are the basis for most CPU chips today. For this technology,

Power = Capacitive load x Voltage^2 x clock rate

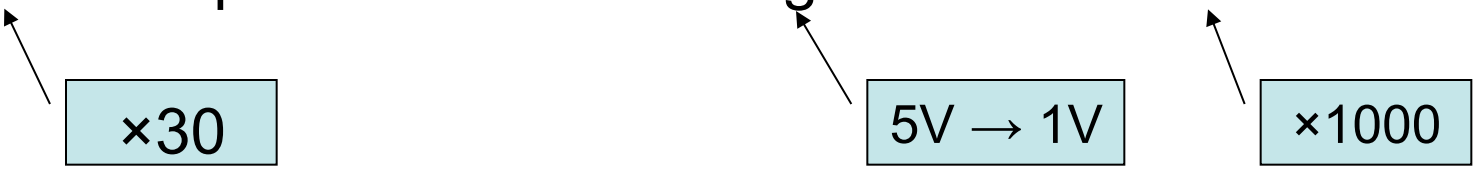


Fig. 1.15:

Capacitive load depends on transistor technology, and transistor *fanout* (no. of inputs an output is connected to)

Only limited reductions for capacitive load and voltage are possible:

- Cannot keep increasing clock rate!

How to increase performance of system?

- Multicore processors

Uniprocessor performance trends (Fig. 1.16)

Summary

Topics covered in this chapter:

- Simple MIPS CPU implementation
- Basic concepts of pipelines
- Performance estimation