LECTURE 5

Single-Cycle
Datapath and Control

PROCESSORS

In lecture 1, we reminded ourselves that the *datapath* and *control* are the two components that come together to be collectively known as the processor.

- Datapath consists of the functional units of the processor.
  - Elements that hold data.
    - Program counter, register file, instruction memory, etc.
  - Elements that operate on data.
    - ALU, adders, etc.
  - Buses for transferring data between elements.
- Control commands the datapath regarding when and how to route and operate on data.

MIPS

To showcase the process of creating a datapath and designing a control, we will be using a subset of the MIPS instruction set. Our available instructions include:

- `add`, `sub`, `and`, `or`, `slt`
- `lw`, `sw`
- `beq`, `j`

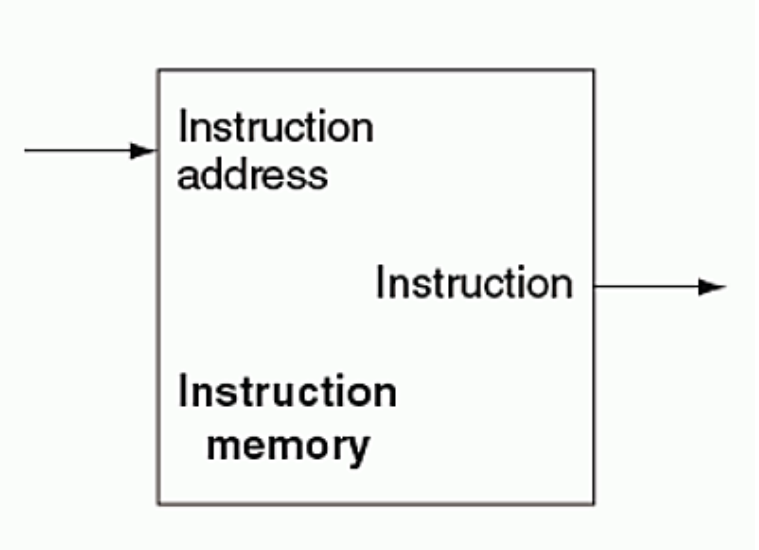
DATAPATH

To start, we will look at the datapath elements needed by every instruction.

First, we have *instruction memory*.

Instruction memory is a state element that provides read-access to the instructions of a program and, given an address as input, supplies the corresponding instruction at that address.

- Code can also be written, e.g., self-modifying code



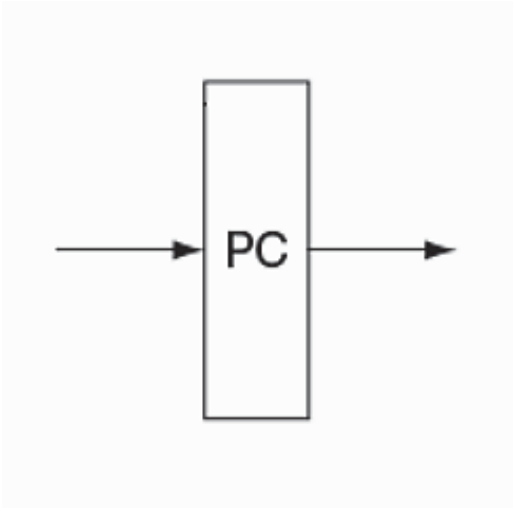
DATAPATH

Next, we have the *program counter* or *PC*.

The PC is a state element that holds the address of the current instruction. Essentially, it is just a 32-bit register which holds the instruction address and is updated at the end of every clock cycle.

- Normally PC increments sequentially except for branch instructions

The arrows on either side indicate that the PC state element is both readable and writeable.

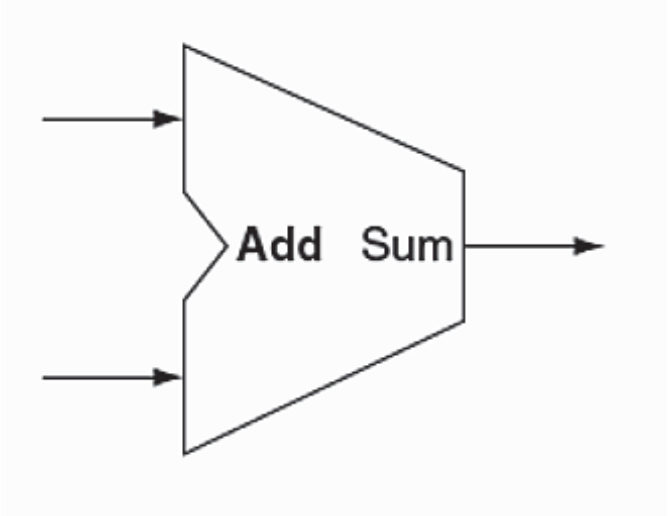


DATAPATH

Lastly, we have the *adder*.

The adder is responsible for incrementing the PC to hold the address of the next instruction.

It takes two input values, adds them together and outputs the result.



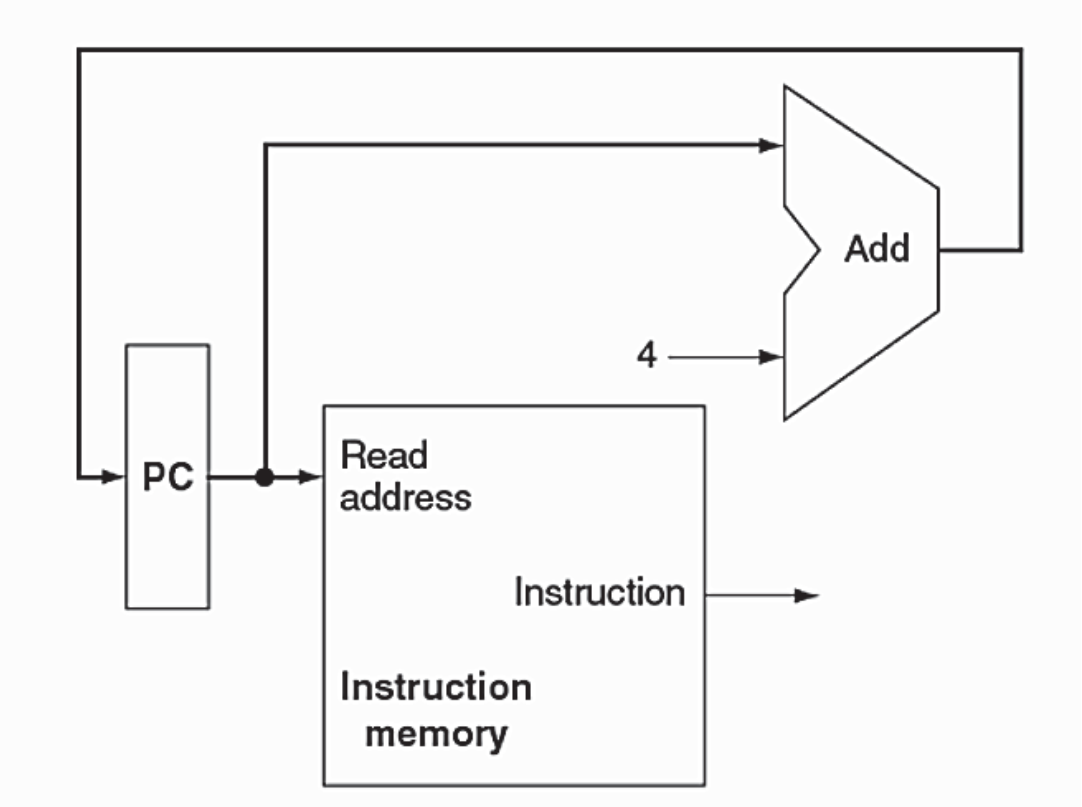
DATAPATH

So now we have instruction memory, PC, and adder datapath elements. Now, we can talk about the general steps taken to execute a program.

- Instruction fetching: use the address in the PC to fetch the current instruction from instruction memory.
- Instruction decoding: determine the fields within the instruction
- Instruction execution: perform the operation indicated by the instruction.
- Update the PC to hold the address of the next instruction.

DATAPATH

- Fetch the instruction at the address in PC.
- Decode the instruction.
- Execute the instruction.
- Update the PC to hold the address of the next instruction.



Note: we perform PC+4 because MIPS instructions are word-aligned.

R-FORMAT INSTRUCTIONS

Now, let's consider R-format instructions. In our limited MIPS instruction set, these are *add*, *sub*, *and*, *or*, and *slt*.

All R-format instructions read two registers, *rs* and *rt*, and write to a register *rd*.

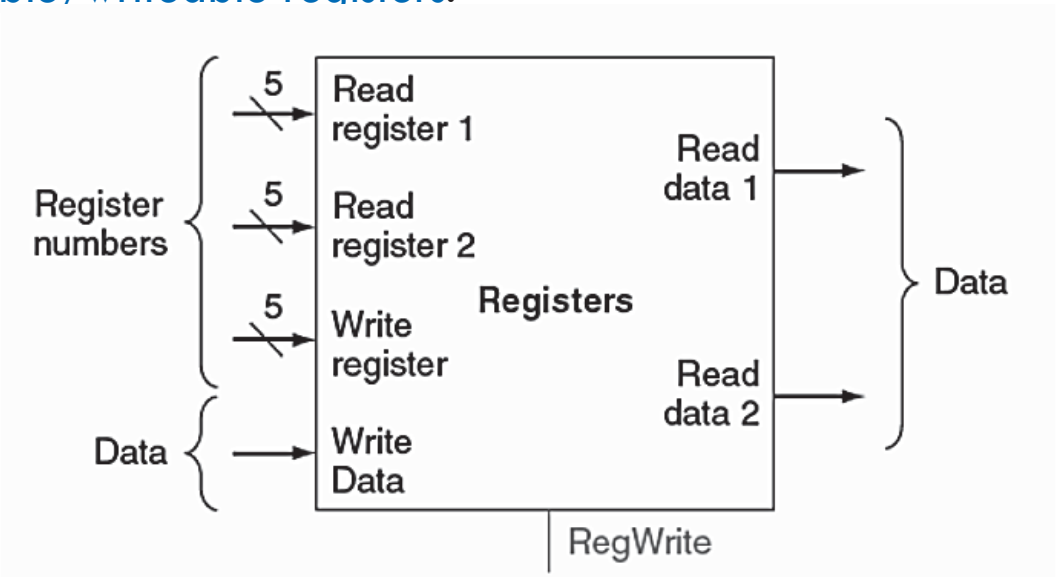
Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R format	op	rs	rt	rd	shamt	funct

op – instruction opcode.
rs – first register source operand.
rt – second register source operand.
rd – register destination operand.
shamt – shift amount.
funct – additional opcodes.

DATAPATH

To support R-format instructions, we'll need to add a state element called a *register file*. A register file is a collection readable/writeable registers.

- `Read register 1` – first source register. 5 bits wide.
- `Read register 2` – second source register. 5 bits wide.
- `Write register` – destination register. 5 bits wide.
- `Write data` – data to be written to a register. 32 bits wide.

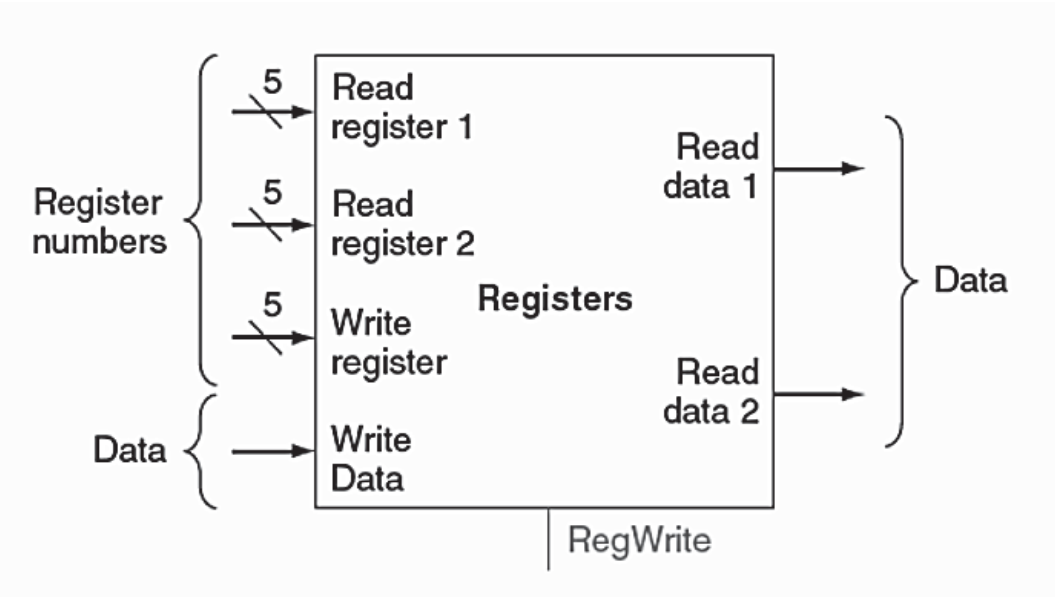


DATAPATH

At the bottom, we have the `RegWrite` input. A writing operation only occurs when this bit is set.

The two output ports are:

- `Read data 1` – contents of source register 1.
- `Read data 2` – contents of source register 2.



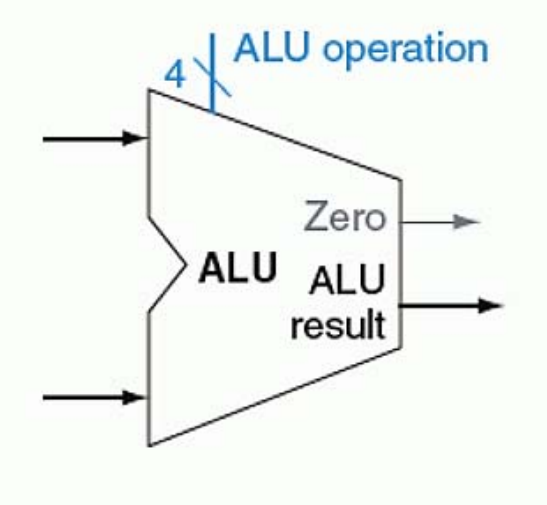
DATAPATH

To actually execute R-format instructions, we need to include the ALU element.

The ALU performs the operation indicated by the instruction. It takes two operands, as well as a 4-bit wide operation selector value. The result of the operation is the output value.

- ALU operation is a part of the control. We discuss datapath first.

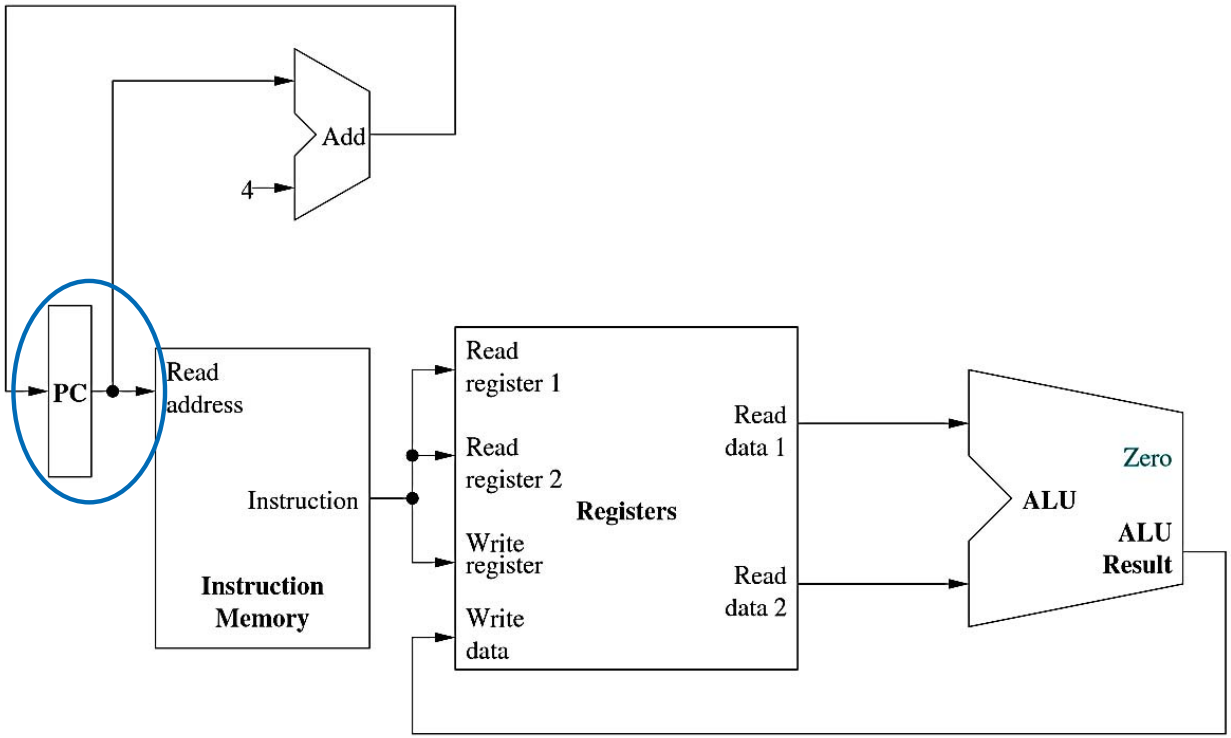
We have an additional output specifically for branching – we will cover this in a minute.



DATAPATH

Here is our datapath for R-format instructions.

1. Grab instruction address from PC.

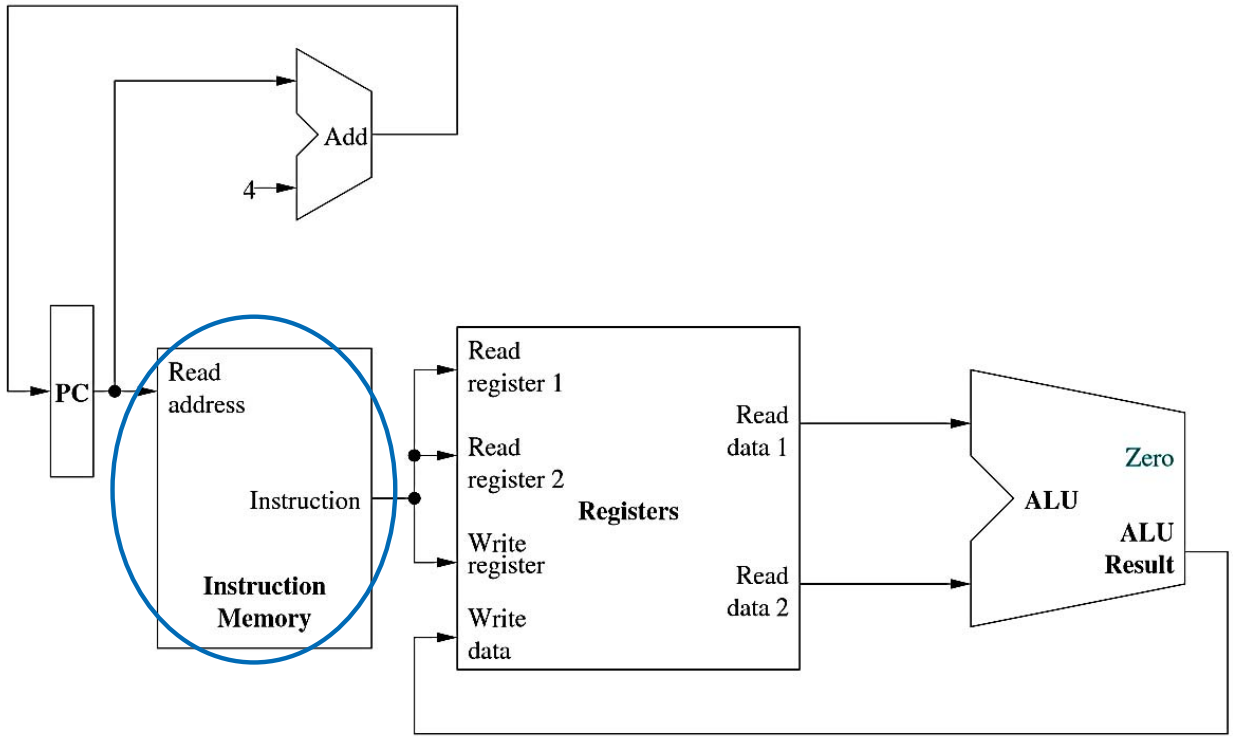


DATAPATH

Here is our datapath for R-format instructions.

2. Fetch instruction from instruction memory.

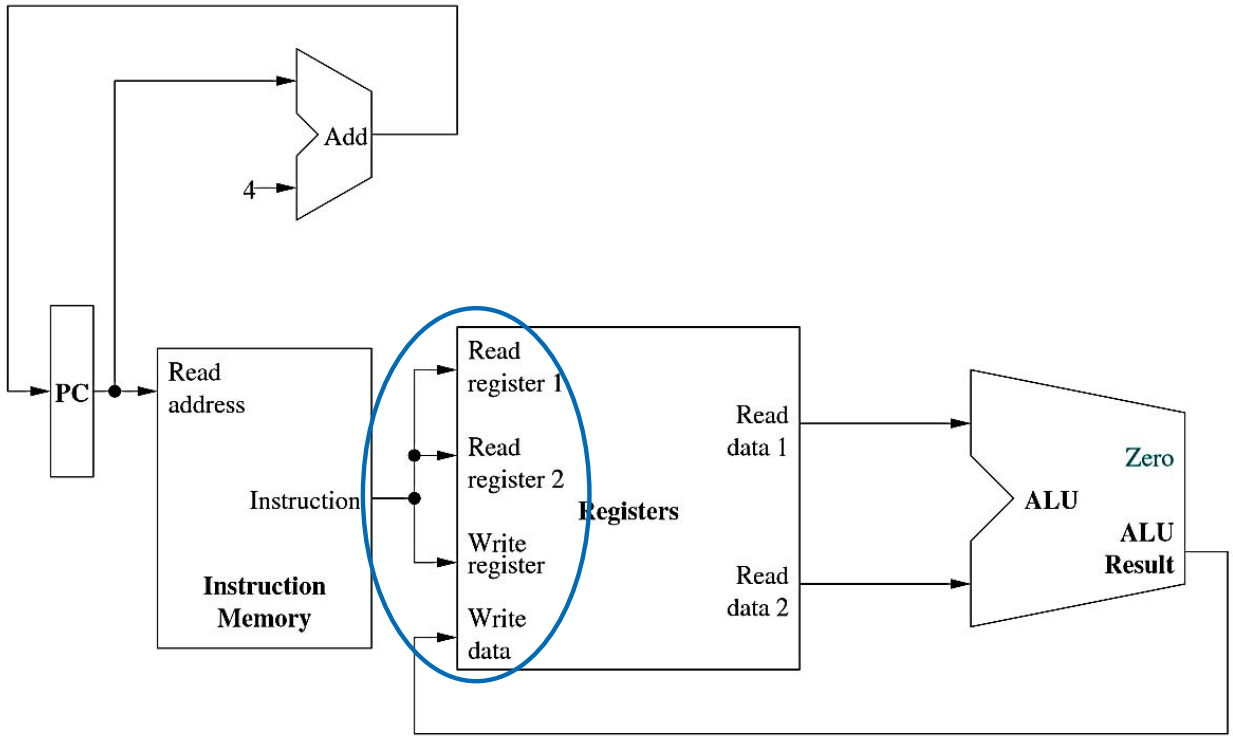
3. Decode instruction.



DATAPATH

Here is our datapath for R-format instructions.

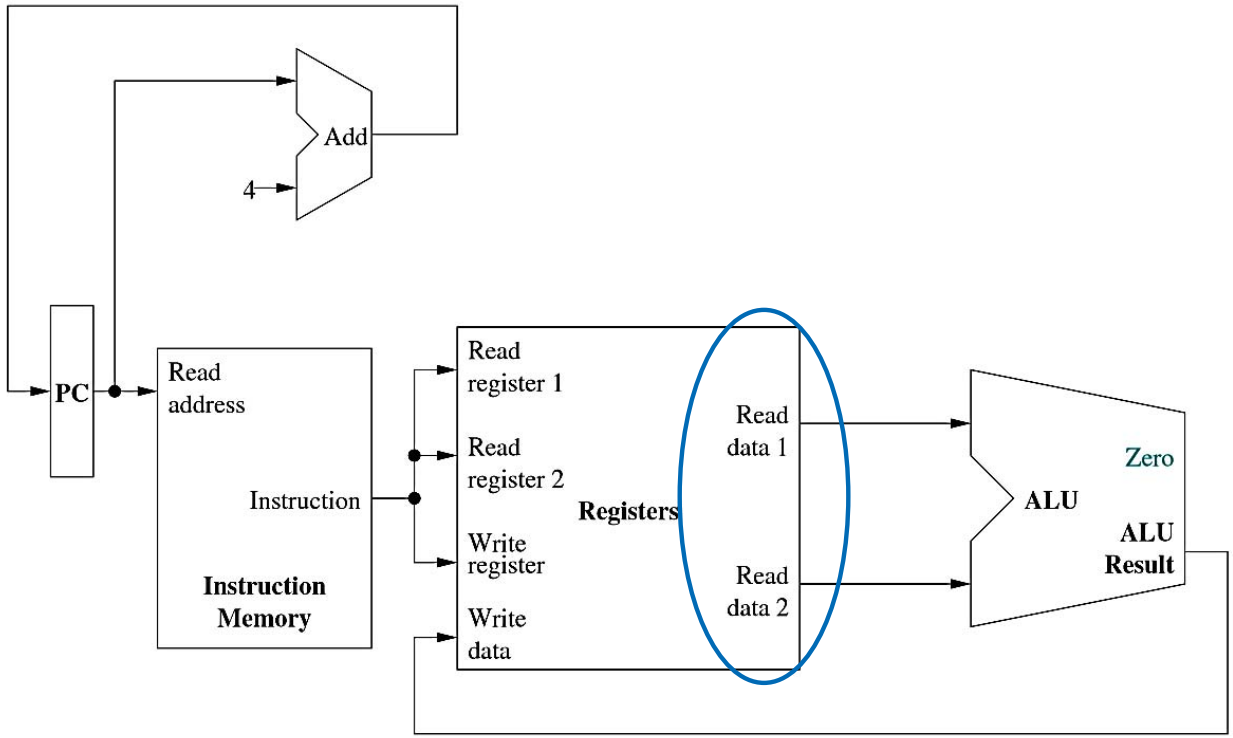
4. Pass `rs`, `rt`, and `rd` into read register and write register arguments.



DATAPATH

Here is our datapath for R-format instructions.

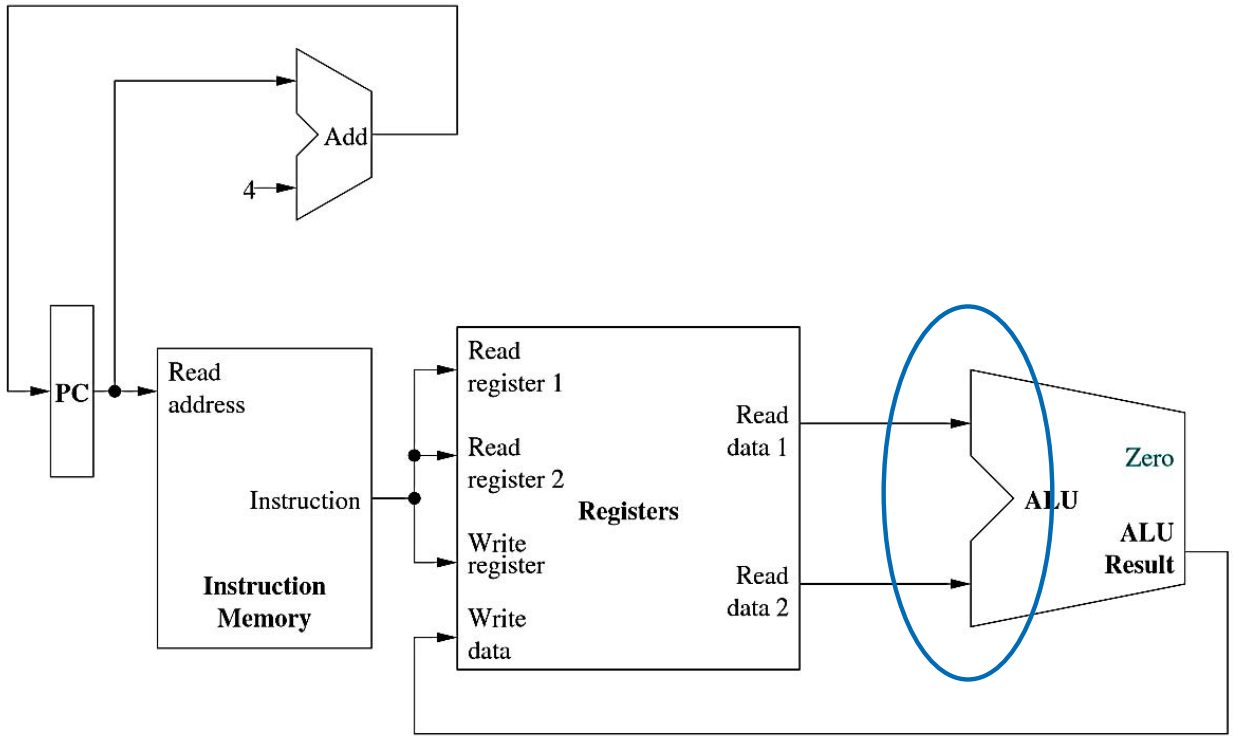
5. Retrieve data from read register 1 and read register 2 (`rs` and `rt`).



DATAPATH

Here is our datapath for R-format instructions.

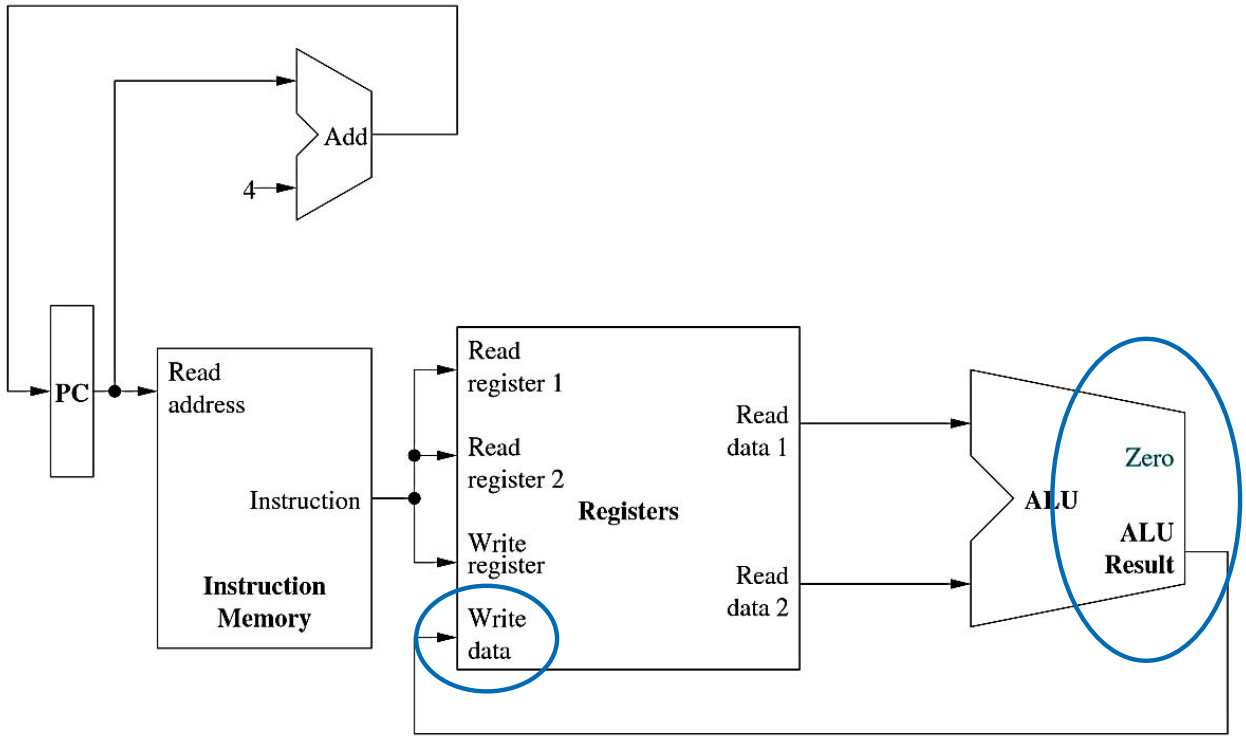
6. Pass contents of `rs` and `rt` into the ALU as operands of the operation to be performed.



DATAPATH

Here is our datapath for R-format instructions.

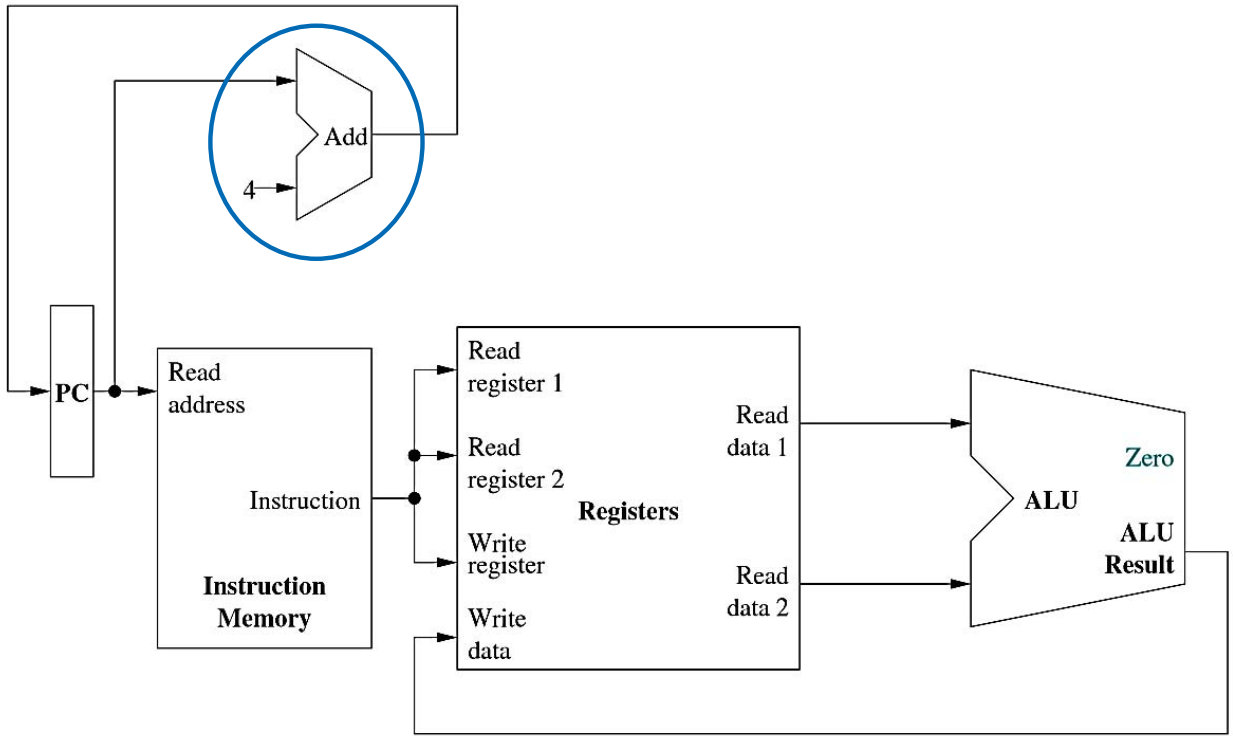
7. Retrieve result of operation performed by ALU and pass back as the write data argument of the register file (with the `RegWrite` bit set).



DATAPATH

Here is our datapath for R-format instructions.

8. Add 4 bytes to the PC value to obtain the word-aligned address of the next instruction.



I-FORMAT INSTRUCTIONS

Now that we have a complete datapath for R-format instructions, let's add in support for I-format instructions. In our limited MIPS instruction set, these are `lw`, `sw`, and `beq`.

- The *op* field is used to identify the type of instruction.
- The *rs* field is the source register.
- The *rt* field is either the source or destination register, depending on the instruction.
- The *immed* field is zero-extended if it is a logical operation. Otherwise, it is sign-extended.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

DATA TRANSFER INSTRUCTIONS

Let's start with accommodating the data transfer. For `lw` and `sw`, we have the following format:

```
lw $rt, immed($rs)
sw $rt, immed($rs)
```

- The memory address is computed by *sign-extending* the 16-bit immediate to 32-bits, which is added to the contents of $rs.
- In lw, $rt represents the register that will be assigned the memory value. In sw, $rt represents the register whose value will be stored in memory.

Bottom line: we need two more datapath elements to access memory and perform sign-extending.

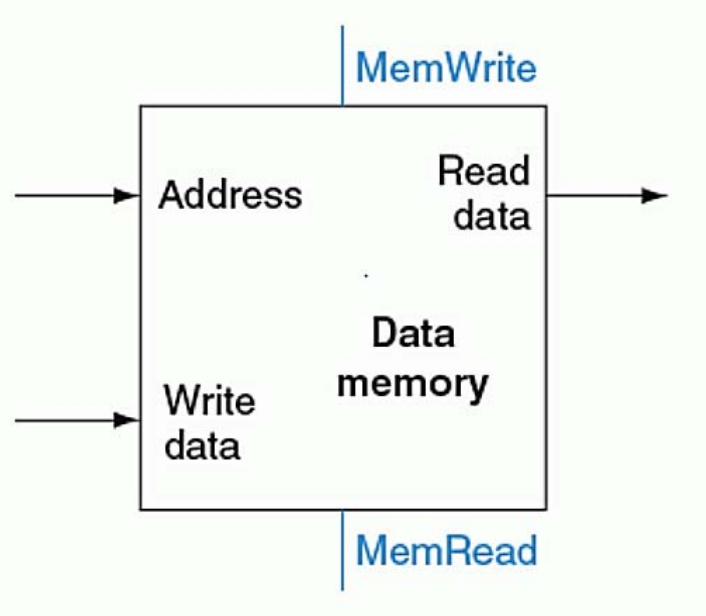
DATAPATH

The *data memory* element implements the functionality for reading and writing data to/from memory.

There are two inputs. One for the address of the memory location to access, the other for the data to be written to memory if applicable.

The output is the data read from the memory location accessed, if applicable.

Reads and writes are signaled by MemRead and MemWrite, respectively, which must be asserted for the corresponding action to take place.

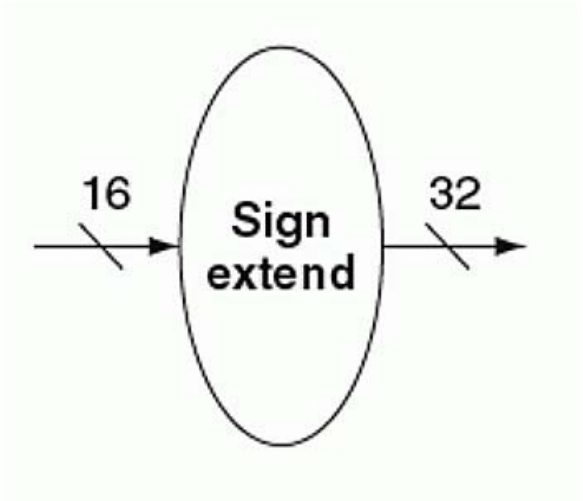


DATAPATH

To perform sign-extending, we can add a sign extension element.

The sign extension element takes as input a 16-bit wide value to be extended to 32-bits.

To sign extend, we simply replicate the most-significant bit of the original field until we have reached the desired field width.

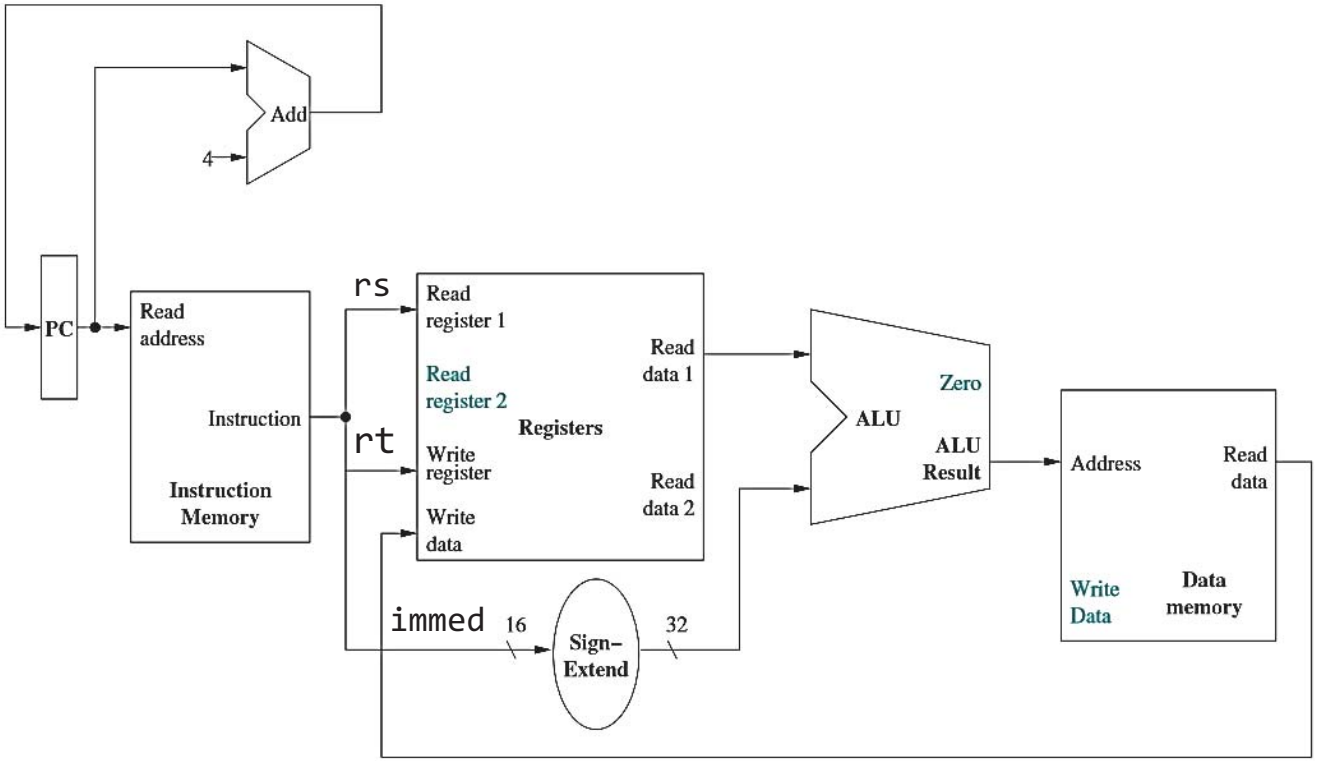


DATAPATH FOR LOAD WORD

Here, we have modified the datapath to work only for the `lw` instruction.

```
lw $rt, immed($rs)
```

The registers have been added to the datapath for added clarity.

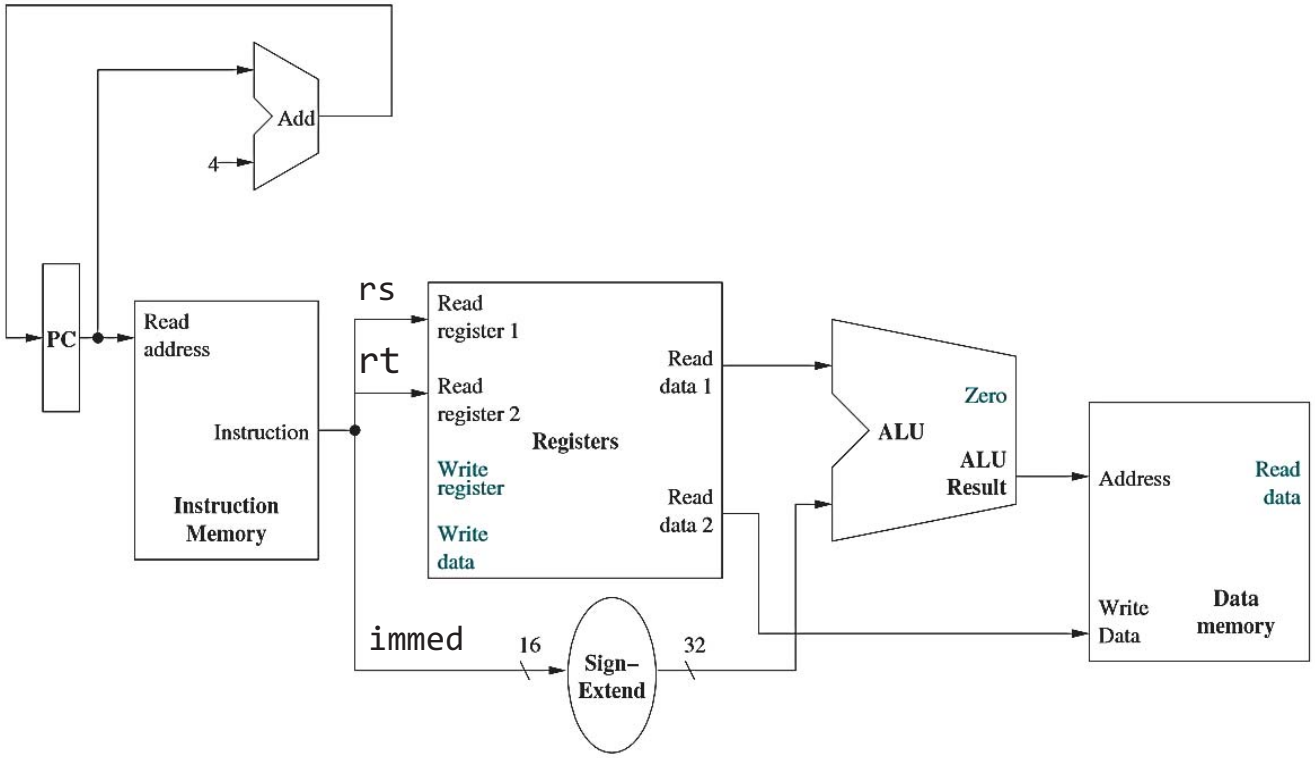


DATAPATH FOR STORE WORD

Here, we have modified the datapath to work only for the `sw` instruction.

```
sw $rt, immed($rs)
```

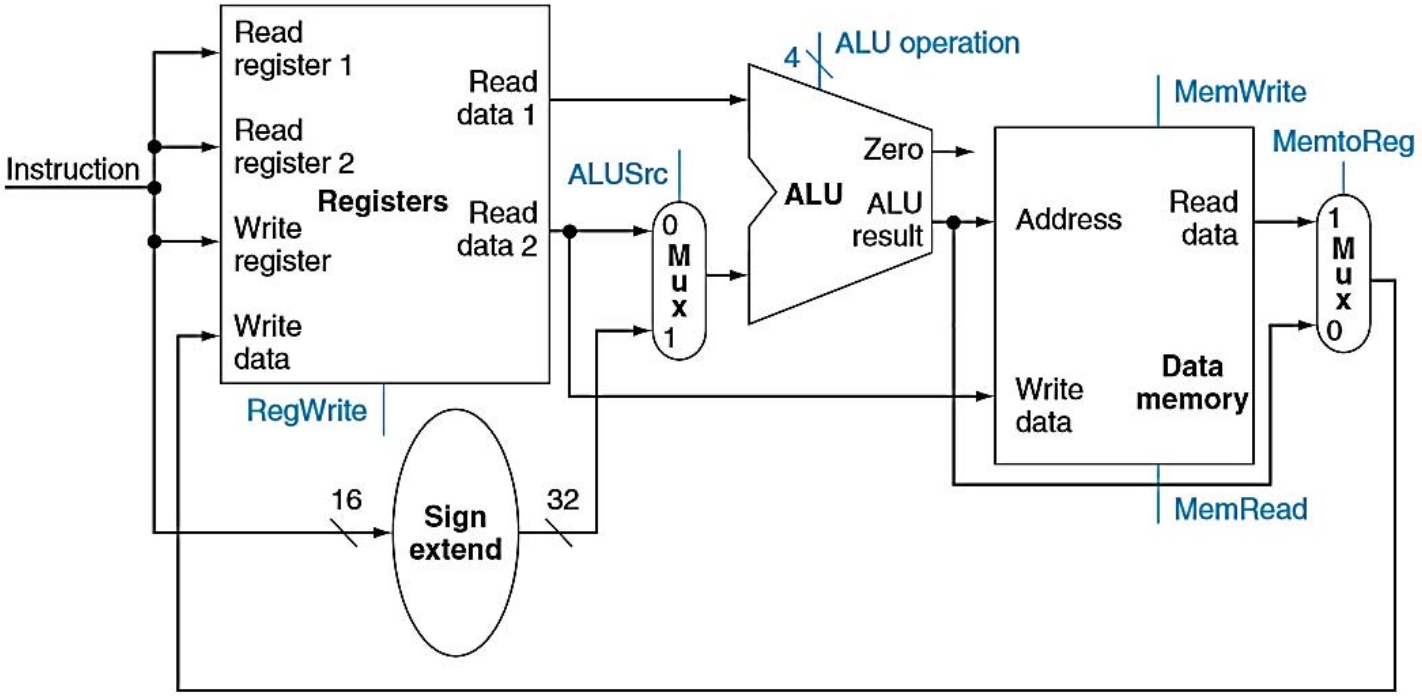
The registers have been added to the datapath for added clarity.



DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

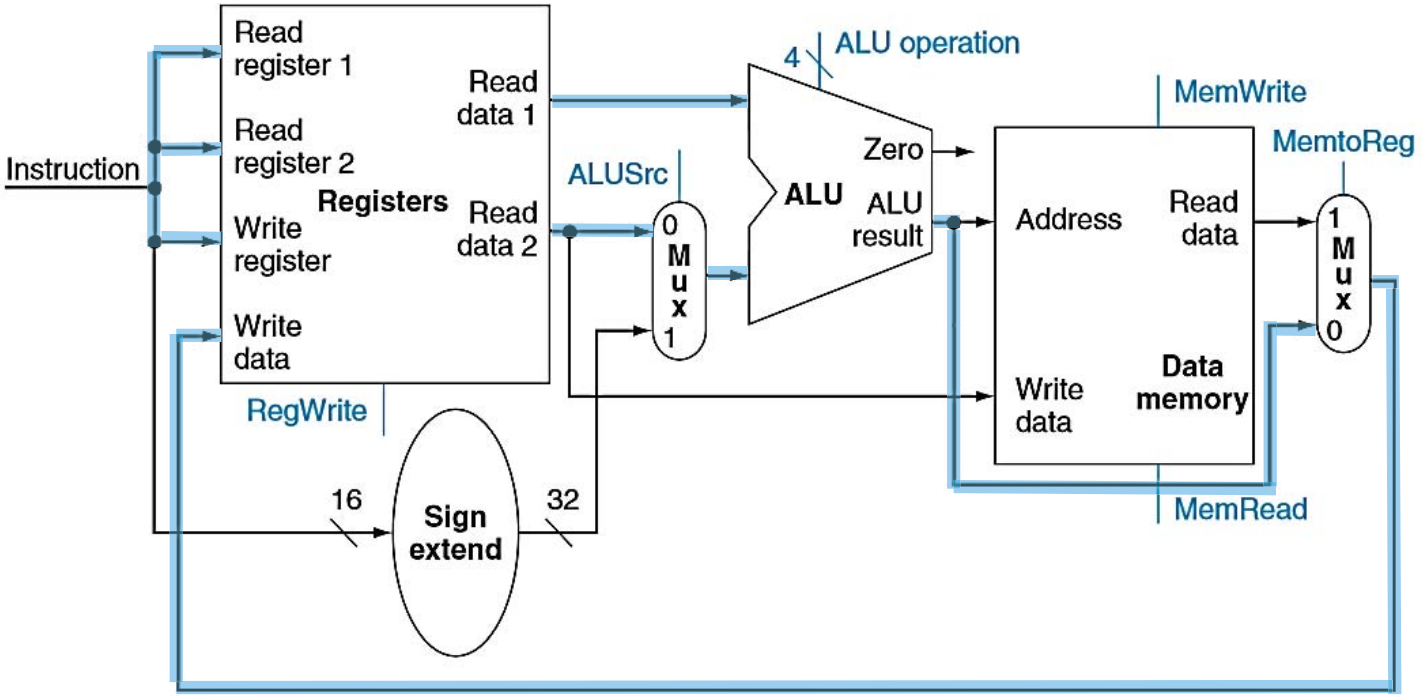
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

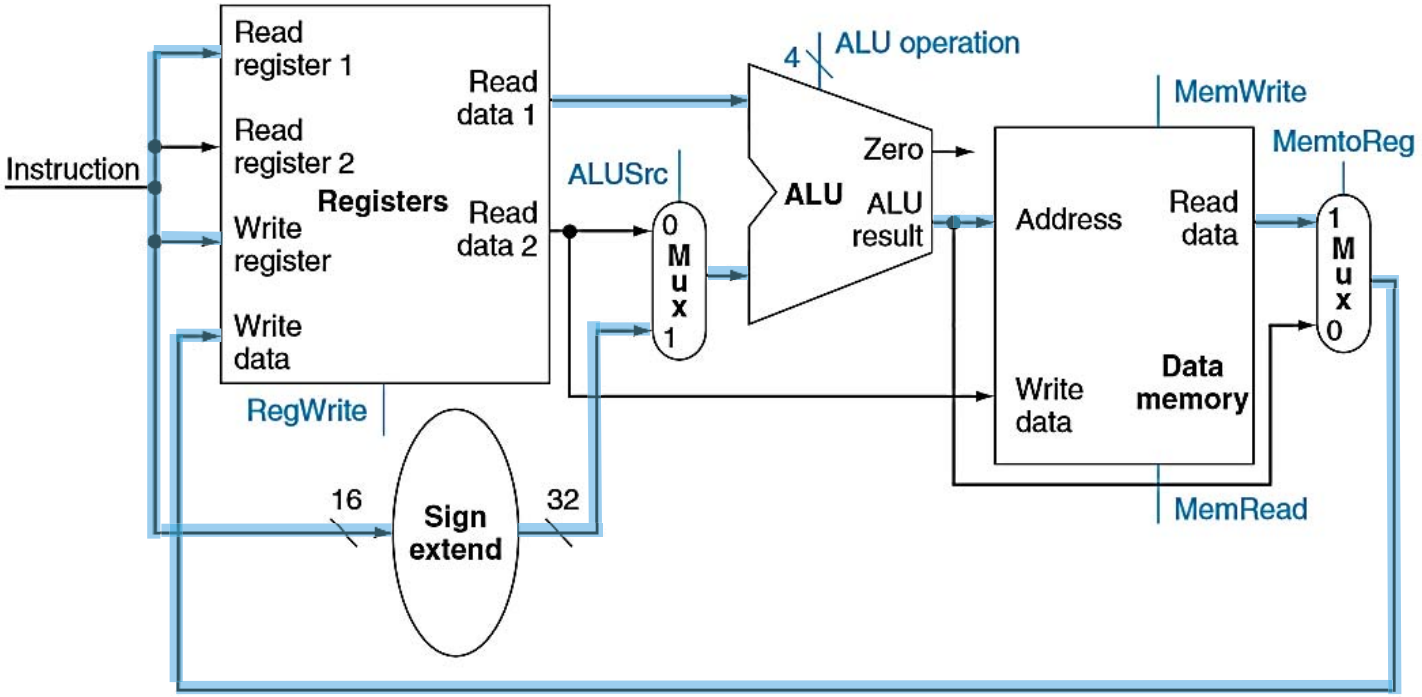
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

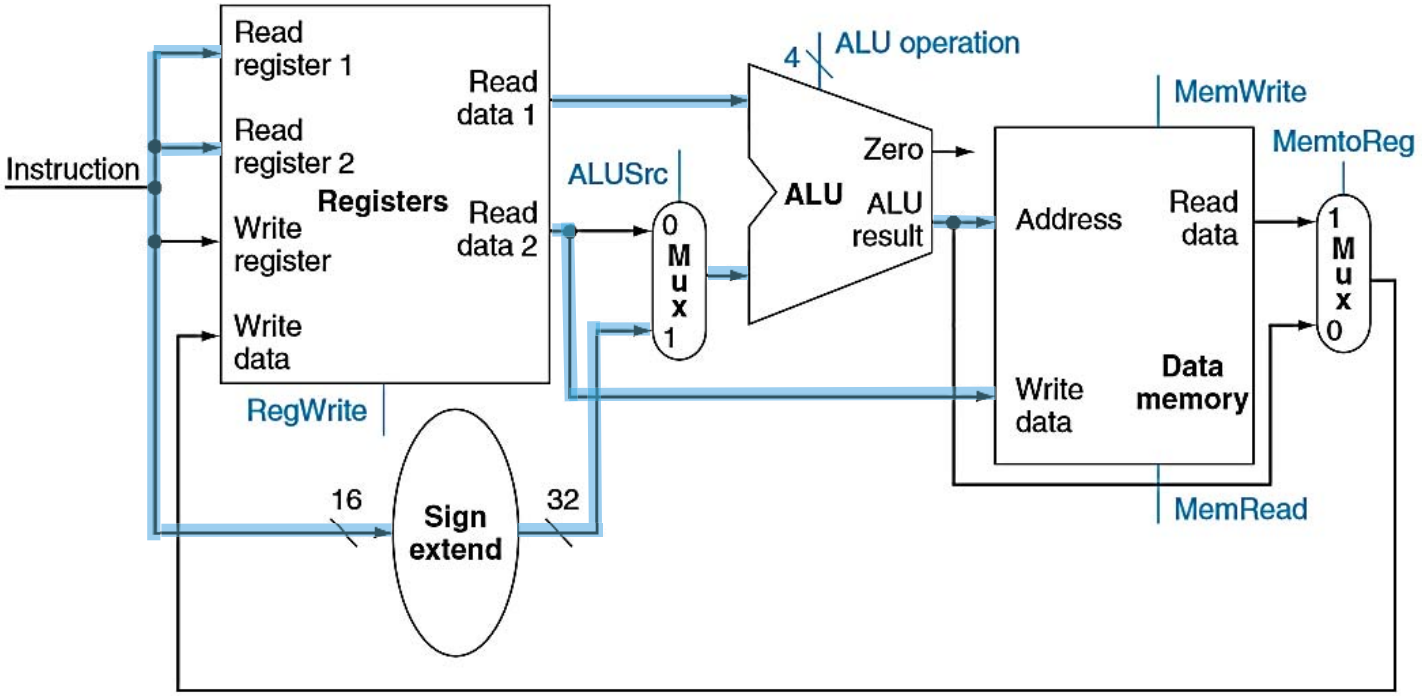
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



DATAPATH FOR R-FORMAT AND MEMORY ACCESS

Note: PC, adder, and instruction memory are omitted.

```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
```



BRANCHING INSTRUCTIONS

Now we'll turn out attention to a branching instruction. In our limited MIPS instruction set, we have the beq instruction which has the following form:

```
beq $t1, $t2, target
```

This instruction compares the contents of $t1 and $t2 for equality and uses the 16-bit immediate field to compute the target address of the branch relative to the current address.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

BRANCHING INSTRUCTIONS

Note that our immediate field is only 16-bits so we can't specify a full 32-bit target address. So we have to do a few things before jumping.

- The immediate field is left-shifted by two because the immediate represents the number of *words* offset from PC+4, not the number of bytes (and we want to get it in number of bytes!).
- We sign-extend the immediate field to 32-bits and add it to PC+4.

```
beq $t1, $t2, target
```

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I format	op	rs	rt	immed		

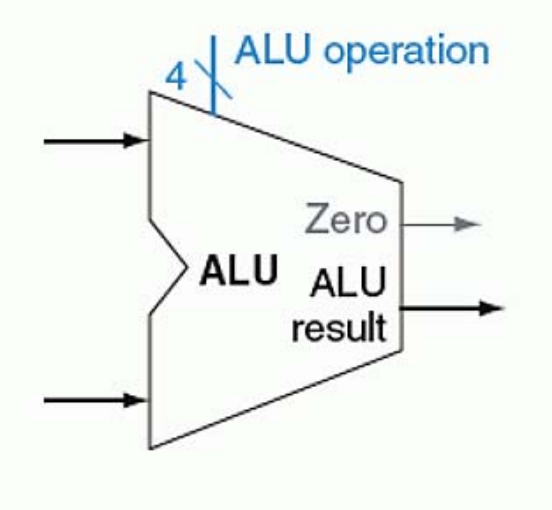
BRANCHING INSTRUCTIONS

Besides computing the target address, a branching instruction also has to compare the contents of the operands.

As stated before, the ALU has an output line denoted as Zero. This output is specifically hardwired to be set when the result of an operation is zero.

To test whether a and b are equal, we can

- set the ALU to perform a subtraction operation.
- The Zero output line is only set if a – b is 0, indicating a and b are equal.

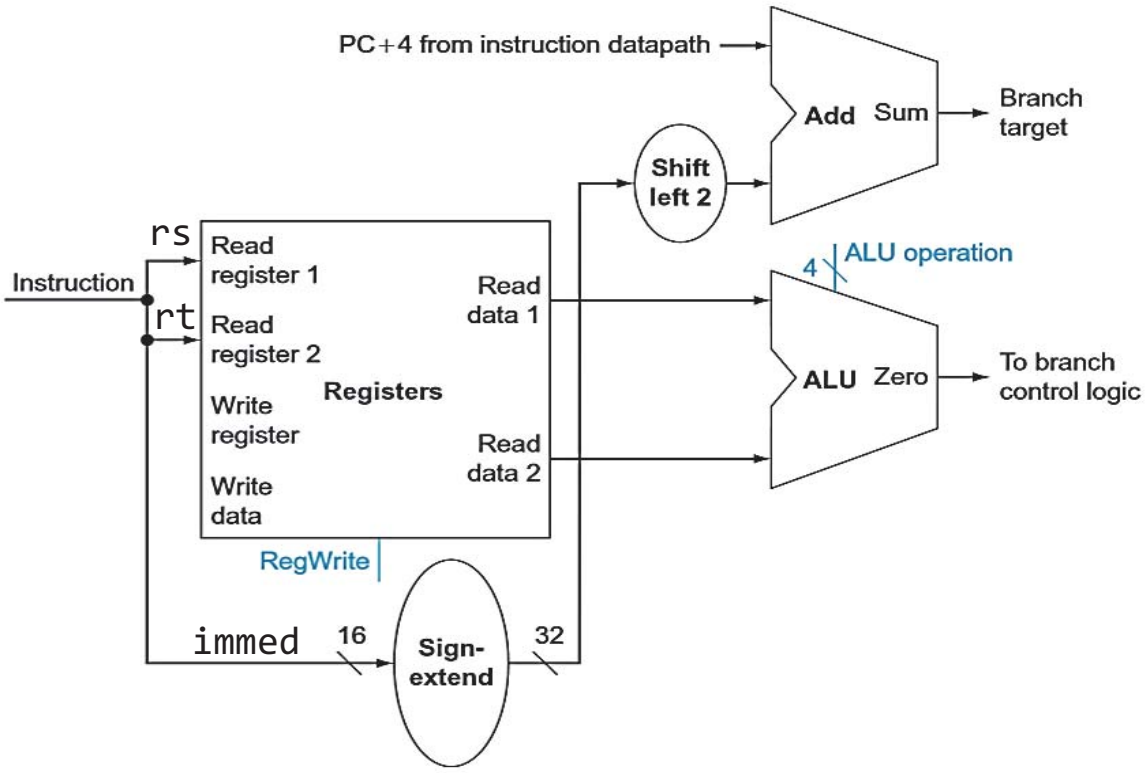


DATAPATH FOR BEQ

Here, we have modified the datapath to work only for the beq instruction.

```
beq $rs, $rt, immed
```

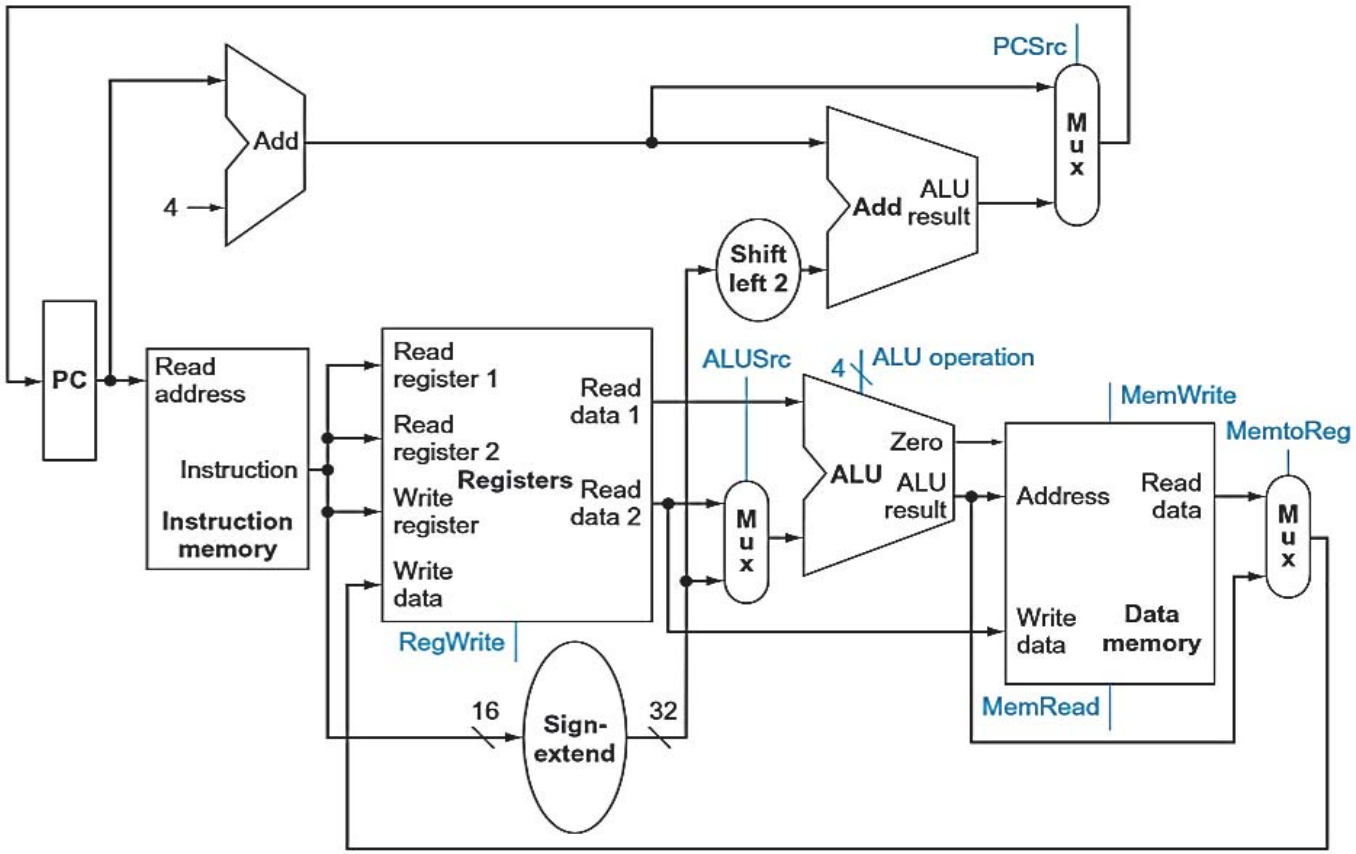
The registers have been added to the datapath for added clarity.



DATAPATH FOR R AND I FORMAT

Now we have a datapath which supports all of our R and I format instructions.

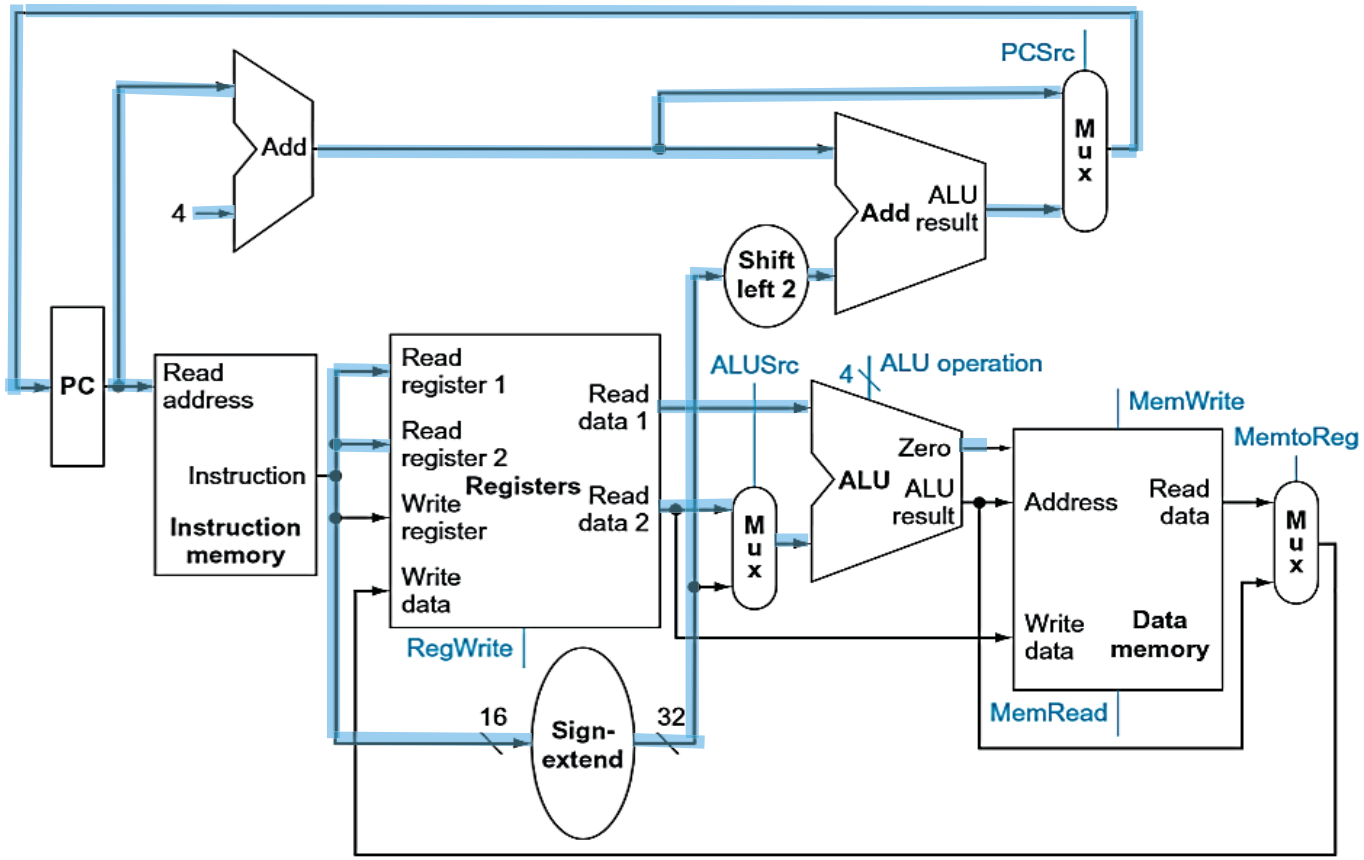
```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
beq $rs, $rt, immed
```



DATAPATH FOR R AND I FORMAT

Now we have a datapath which supports all of our R and I format instructions.

```
add $rd, $rs, $rt
lw $rt, immed($rs)
sw $rt, immed($rs)
beq $rs, $rt, immed
```



J-FORMAT INSTRUCTIONS

The last instruction we have to implement in our simple MIPS subset is the jump instruction. An example jump instruction is `j L1`. This instruction indicates that the next instruction to be executed is at the address of label L1.

- We have 6 bits for the opcode.
- We have 26 bits for the target address.

Name	**Fields**					
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
J format	op	target_addr				

J-FORMAT INSTRUCTIONS

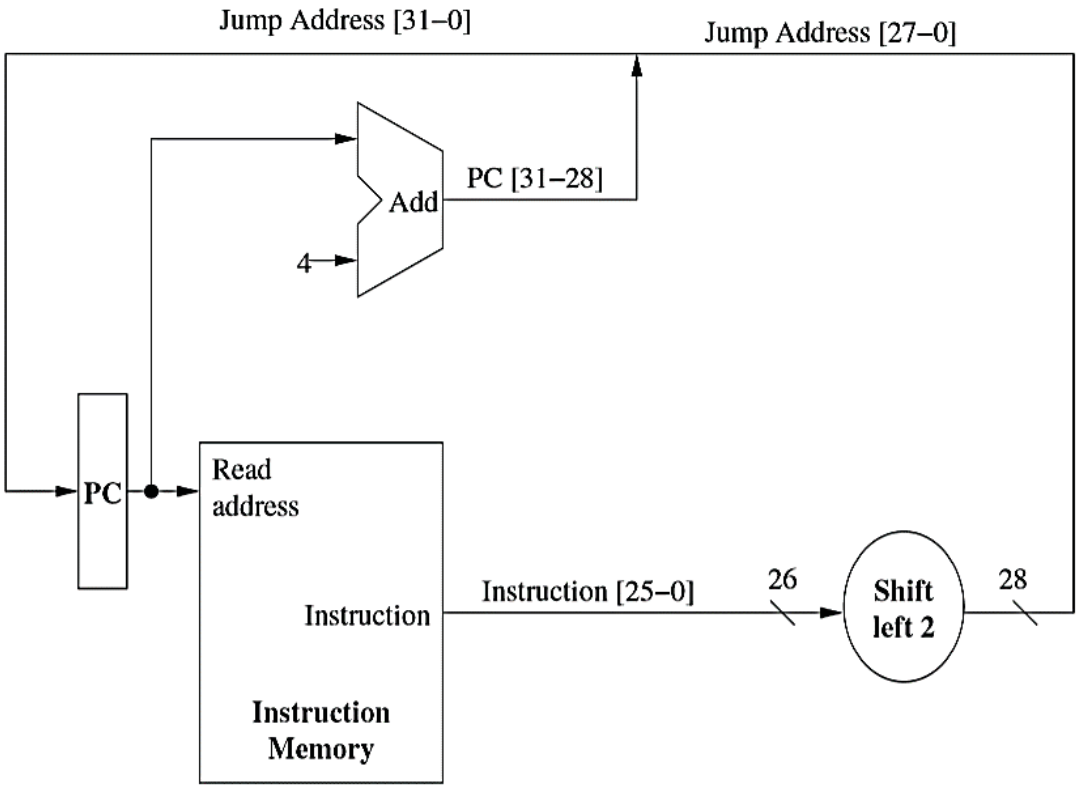
Note, we do not have enough space in the instruction to specify a full target address.

- Branching solves this problem by specifying an offset in words.
- Jump instructions solve this problem by specifying *a portion of an absolute address*
  - Take the 26-bit target address field of the instruction, left-shift by two (instructions are word-aligned),
  - concatenate the result with the upper 4 bits of PC+4.

DATAPATH FOR J-FORMAT

Here, we have modified the datapath to work only for the `j` instruction.

```
j targaddr
```



SINGLE-CYCLE CONTROL

Now we have a complete datapath for our simple MIPS subset – we will show the whole diagram in just a couple of minutes. Before that, we will add the control.

The *control unit* is responsible for taking the instruction and generating the appropriate signals for the datapath elements.

Signals that need to be generated include

- Operation to be performed by ALU.
- Whether register file needs to be written.
- Signals for multiple intermediate multiplexors.
- Whether data memory needs to be written.

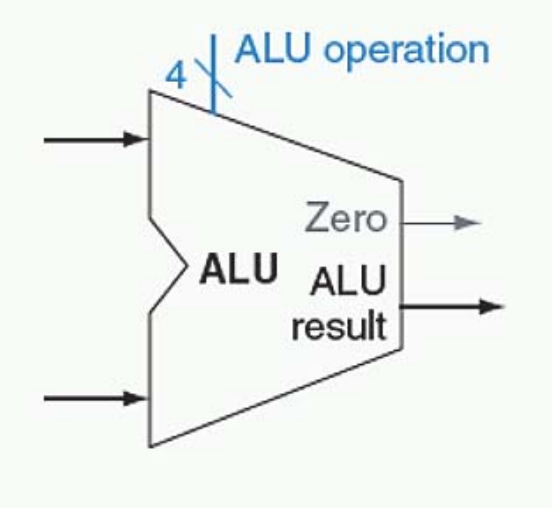
For the most part, we can generate these signals using only the *opcode* and *funct* fields of an instruction.

ALU CONTROL LINES

Note here that the ALU has a 4-bit control line called ALU operation. The first two bits indicate whether a and b need to be inverted, respectively. The last two bits indicate the operation.

- Remember subtract is implemented as add (thus the inversion)

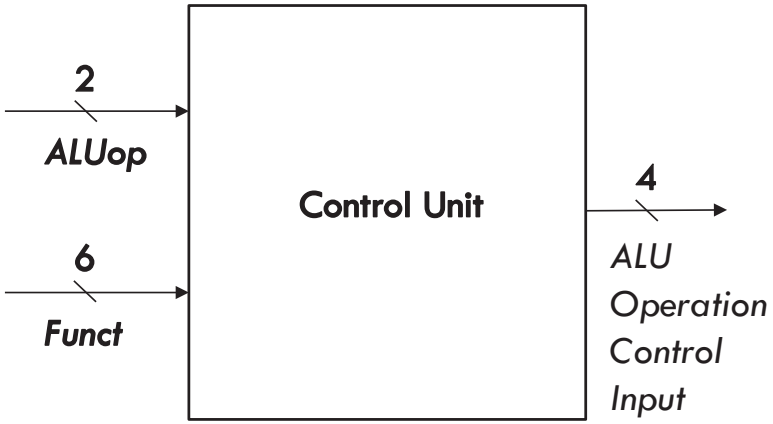
ALU Control Lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR



ALU CONTROL LINES

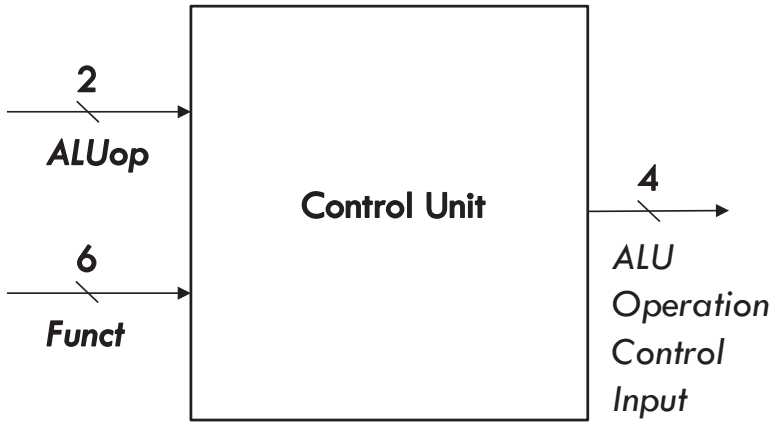
How do we set these control lines? Consider the control unit below.

- The 2-bit *ALUop* input indicates whether an operation should be add (00) *for loads and stores*, subtract (01) *for beq*, or determined by the funct input (10).
  - Case 00 and 01 are determined by the op field
  - Case 10 is used by R-format instructions
- The 6-bit *Funct* input corresponds to the funct field of R-format instructions. Each unique funct field corresponds to a unique set of ALU control input lines.



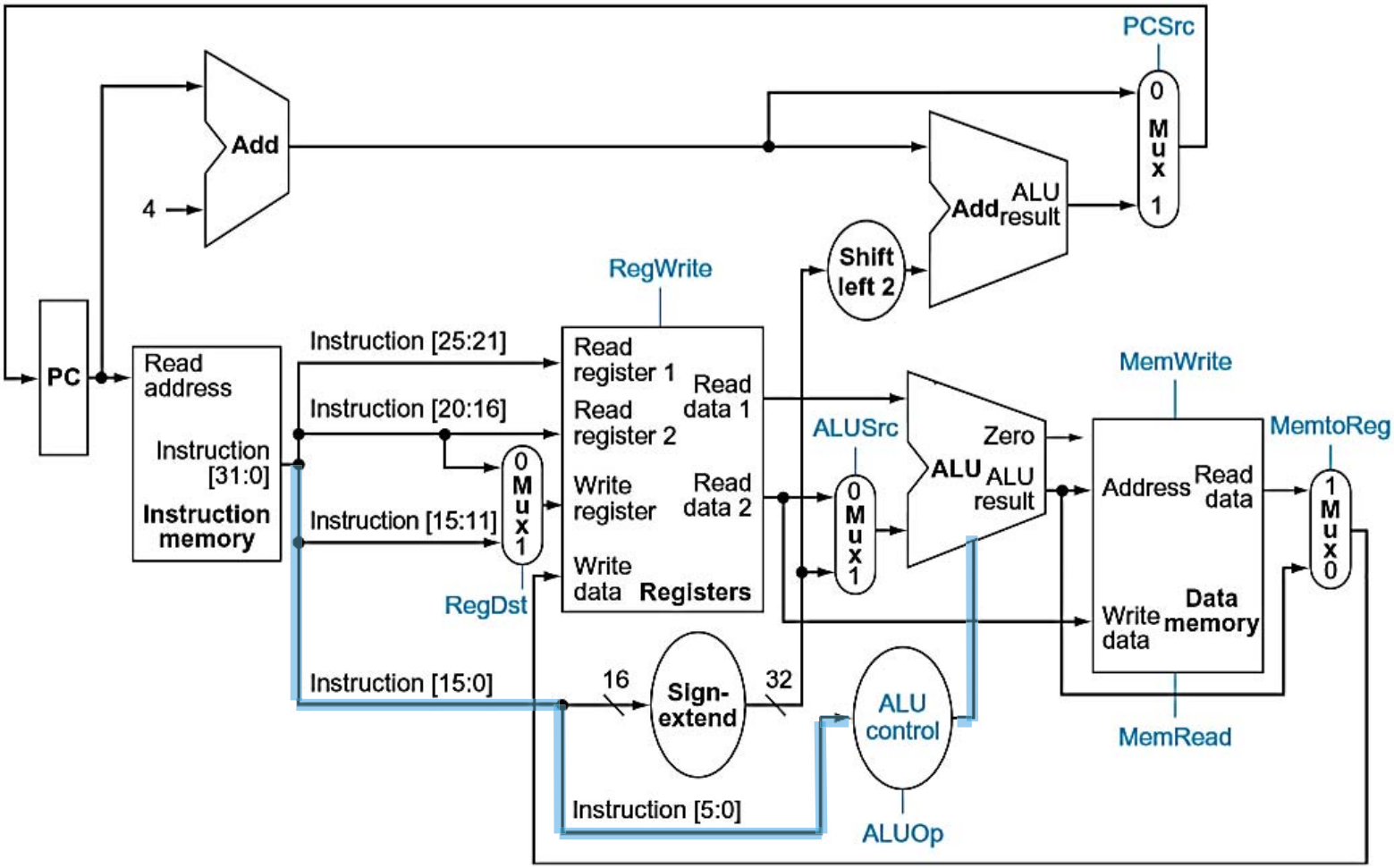
ALU CONTROL LINES

Opcode	ALU op	Operation	Funct	ALU action	ALU Control Input
lw	00	Load word	N/A	add	0010
sw	00	Store word	N/A	add	0010
beq	01	Branch equal	N/A	subtract	0110
R-type	10	Add	100000	add	0010
R-type	10	Subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	Set on less than	101010	slt	0111



Here, we have modified the datapath to work with every instruction *except the jump instruction.*

Notice the added element for determining the ALU control input from the **funct** field for R-types.



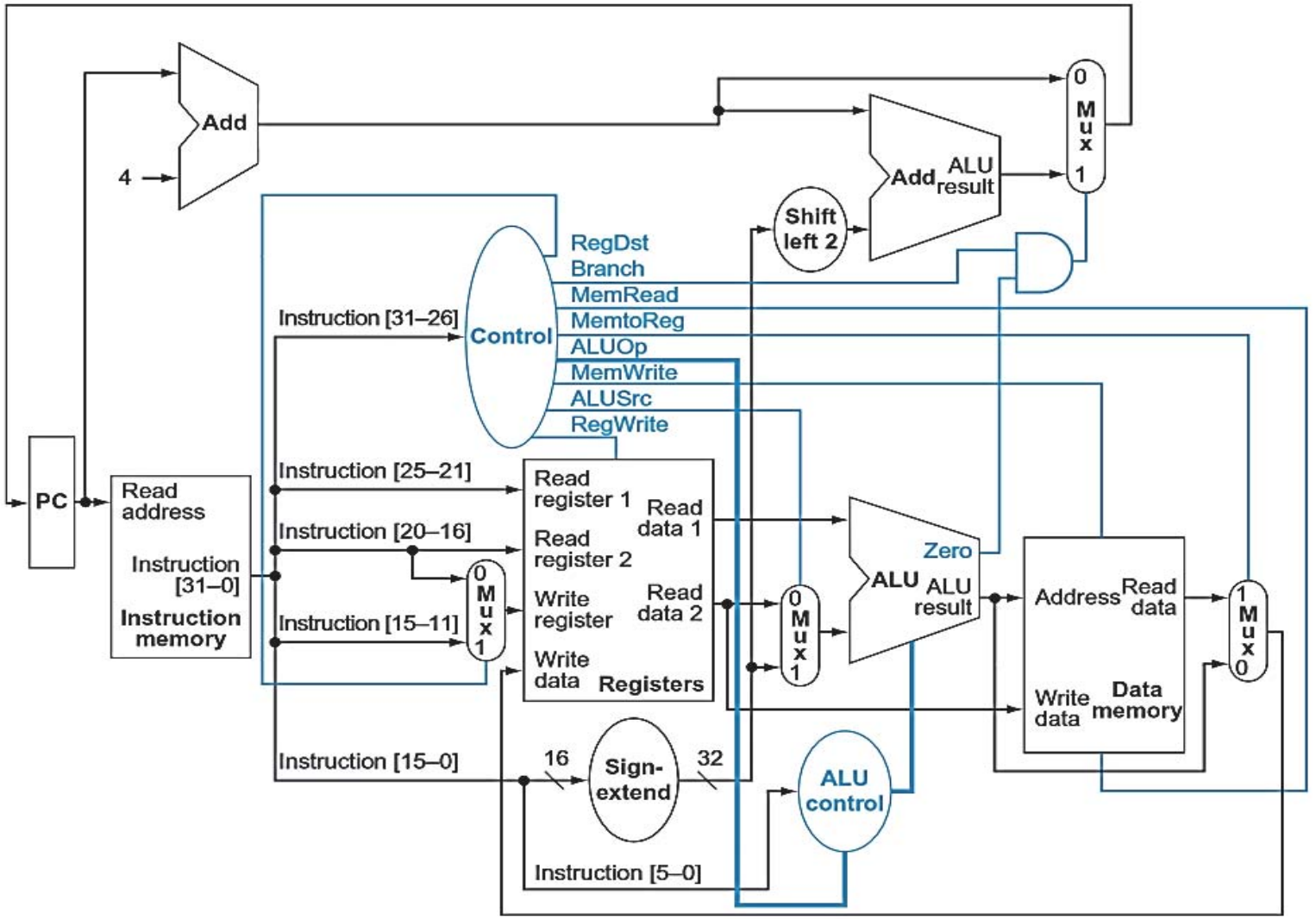
CONTROL SIGNALS

As we can see from the previous slide, we also need to use the instruction to set control signals other than the ALU.

Signal Name	Effect when not set	Effect when set
RegDst	Destination register comes from *rt* field.	Destination register comes from the *rd* field.
RegWrite	None.	Write Register is written to with Write Data.
ALUSrc	Second ALU operand is Read Data 2.	Second ALU operand is immediate field.
PCSrc	PC → PC + 4	PC → Branch target
MemRead	None.	Contents of Address input are copied to Read Data.
MemWrite	None.	Write Data is written to Address.
MemToReg	Value of register Write Data is from ALU.	Value of register Write Data is memory Read Data.

Here, we add the control logic for every instruction except the jump instruction.

Notice how most of the control decisions can be decided using only the upper 6 bits of the instruction (opcode).



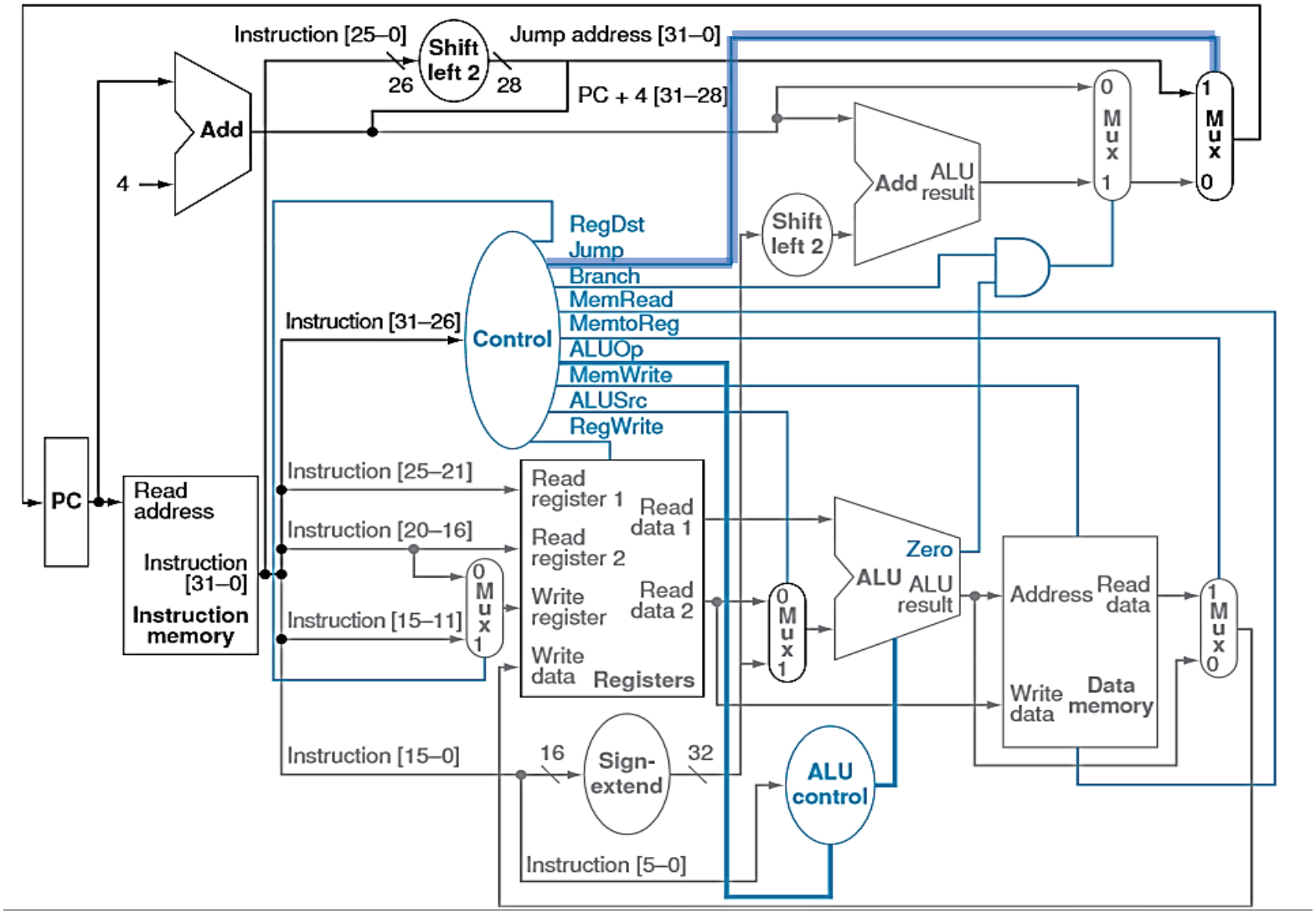
CONTROL SIGNALS

From the previous slide, we can see that the control signals are chosen based on the upper 6 bits of the instruction. That is, the ***opcode*** is used to set the control lines.

Instr.	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
R	1	0	0	1	0	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	0	0	1	0	1

Furthermore, as we saw before, the ALU control input lines are also dictated by the funct fields of applicable instructions.

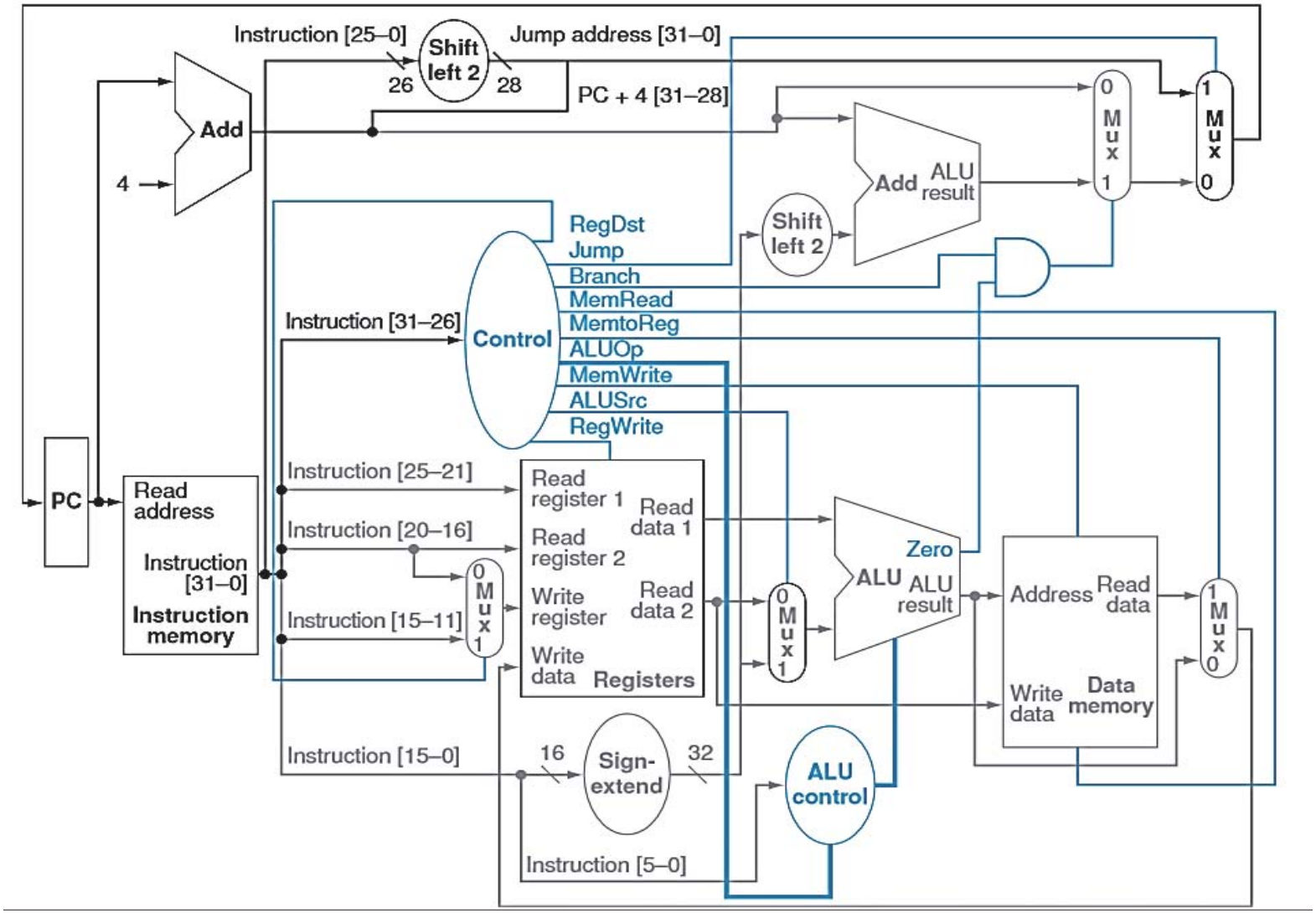
Here we add in an additional control line for jump instructions.



Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

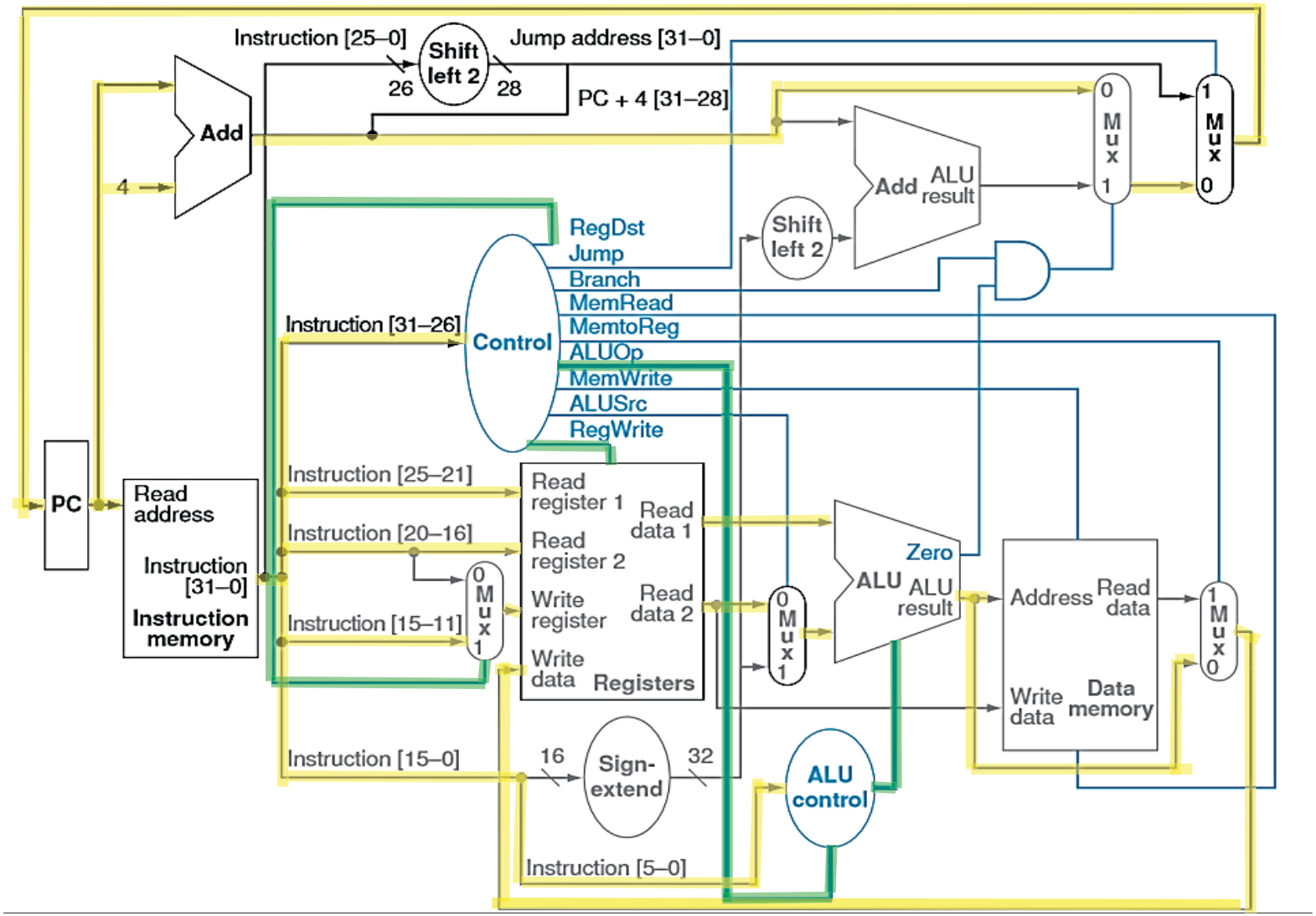


Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

Datapath shown in yellow. Relevant control line assertions in green.

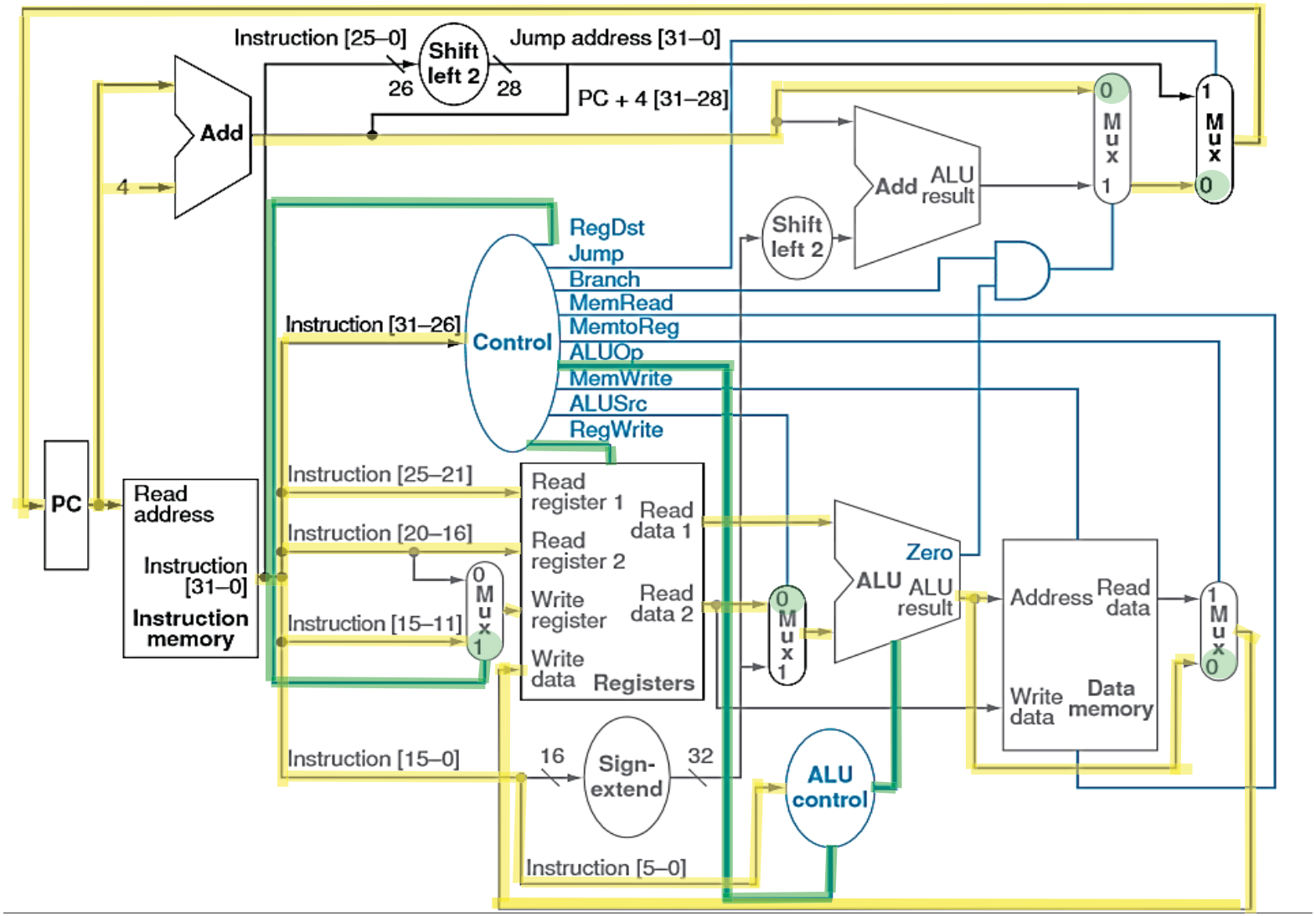


Quiz Time!

What are the relevant datapath lines for the add instruction and what are the values of each of the control lines?

```
add $rd, $rs, $rt
```

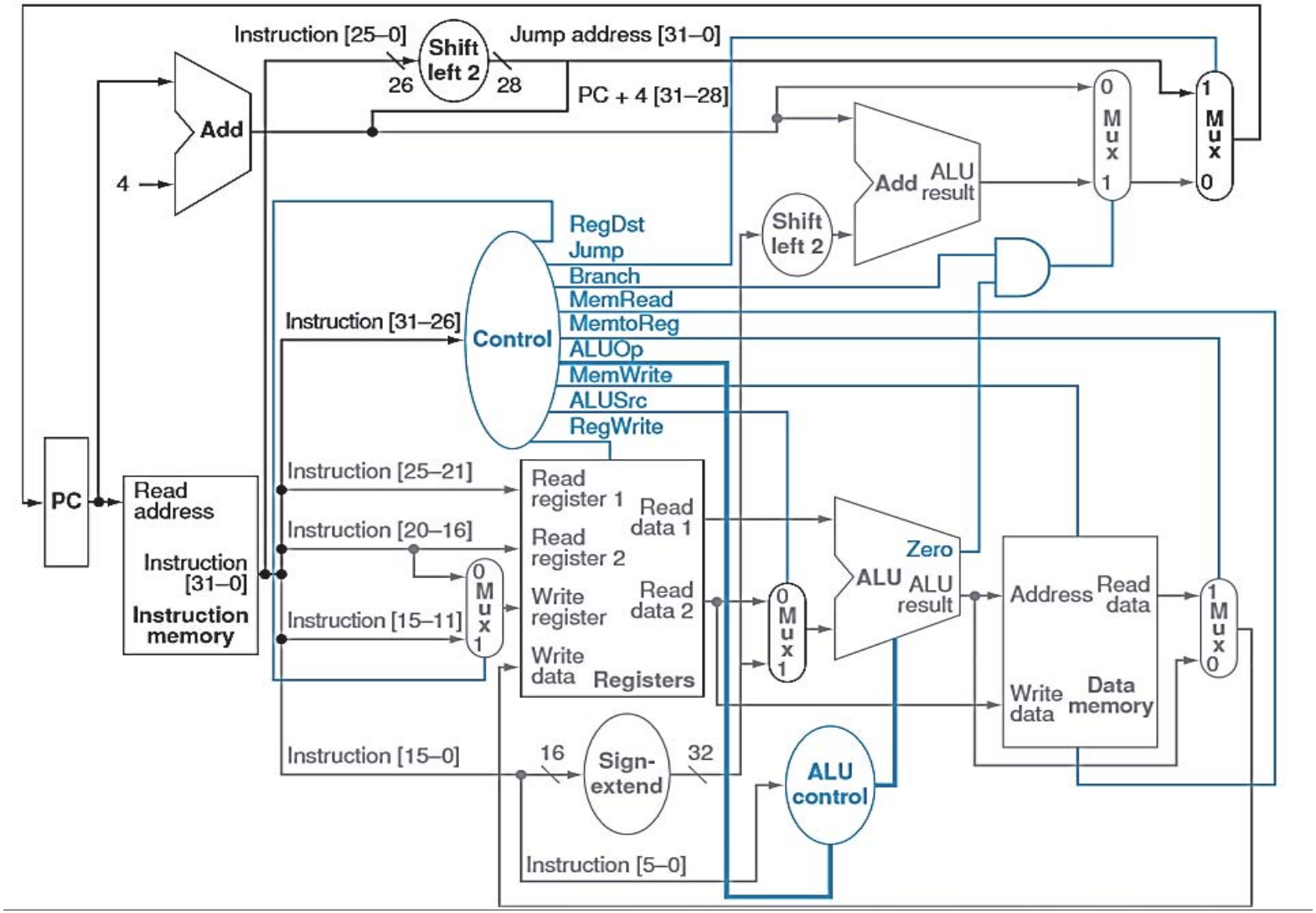
Datapath shown in yellow. Relevant control line assertions in green.



Quiz Time!

What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

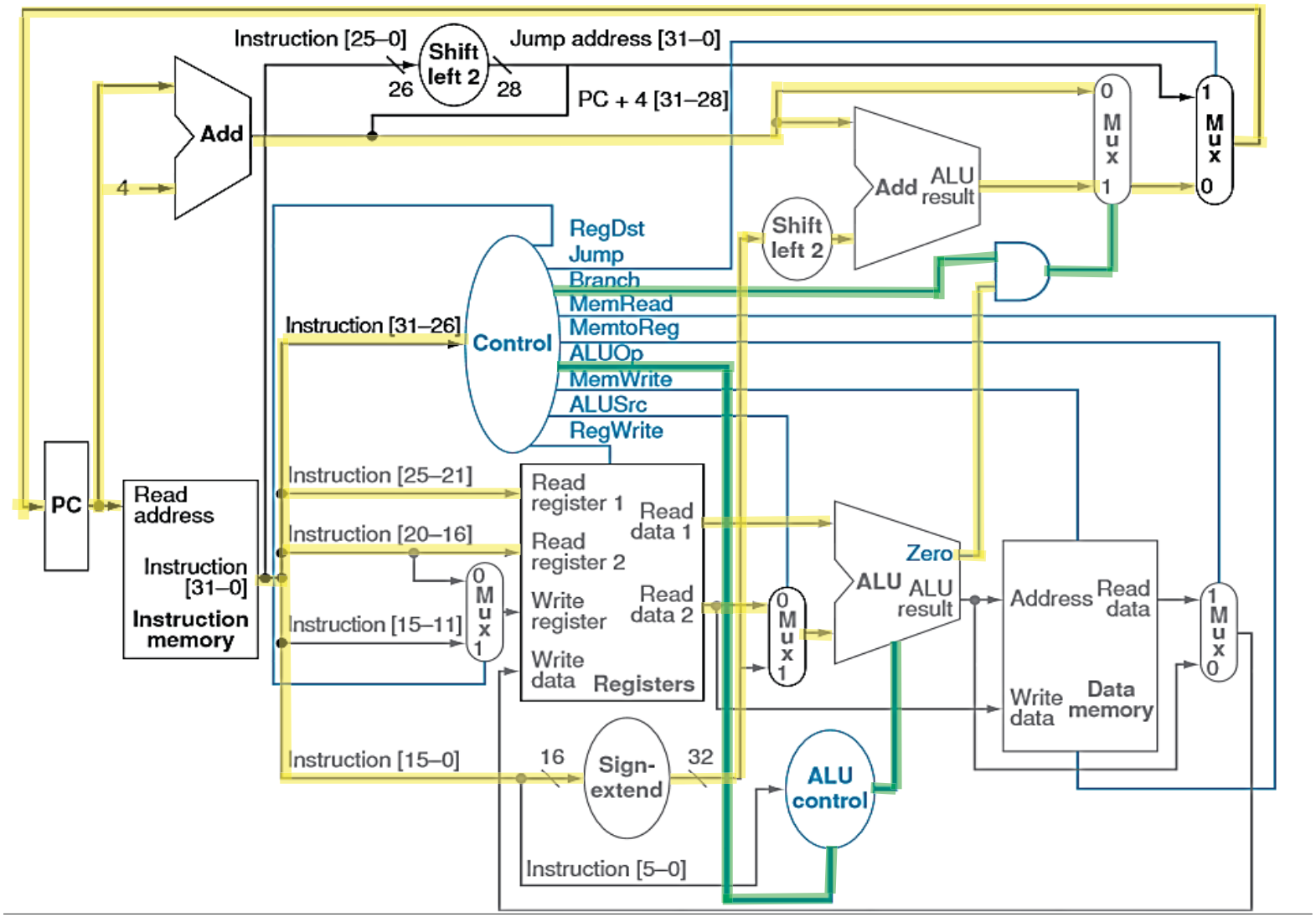
```
beq $rs, $rt, imm
```



Quiz Time!

What are the relevant datapath lines for the beq instruction and what are the values of each of the control lines?

```
beq $rs, $rt, imm
```



RELATIVE CYCLE TIME

What is the longest path (slowest instruction) assuming 4ns for instruction and data memory, 3ns for ALU and adders, and 1ns for register reads or writes? Assume negligible delays for muxes, control unit, sign extend, PC access, shift left by 2, routing, etc

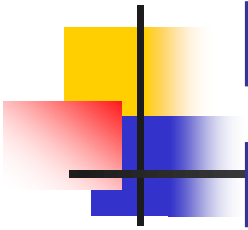
Type	Instruction Memory	Register Read	ALU Operation	Data Memory	Register Write	Total
R-format	4	1	3	0	1	9
lw	4	1	3	4	1	13
sw	4	1	3	4	0	12
beq	4	1	3	0	0	8
j	4	0	0	0	0	4

SINGLE-CYCLE IMPLEMENTATION

The advantage of single cycle implementation is that it is simple to implement.

Disadvantages

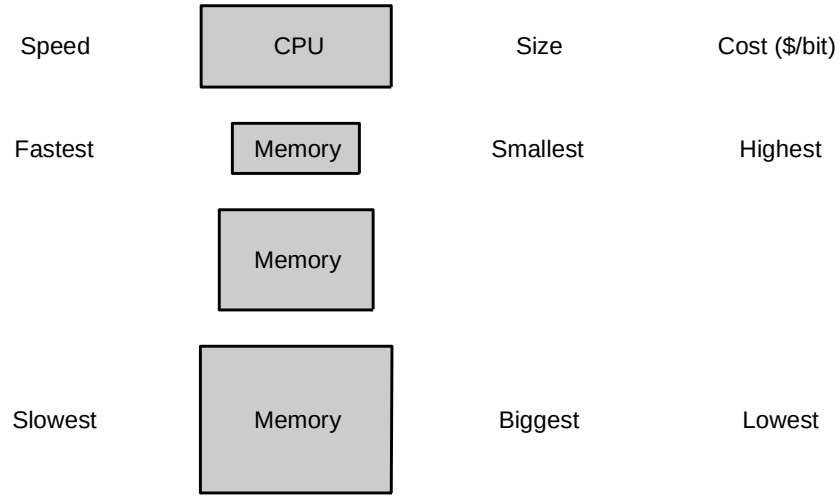
- The clock cycle will be determined by the longest possible path, which is not the most common instruction. This type of implementation violates the idea of making the common case fast.
- May be wasteful with respect to area since some functional units, such as adders, must be duplicated since they cannot be shared during a single clock cycle



COD Ch. 7
Large and Fast: Exploiting Memory Hierarchy

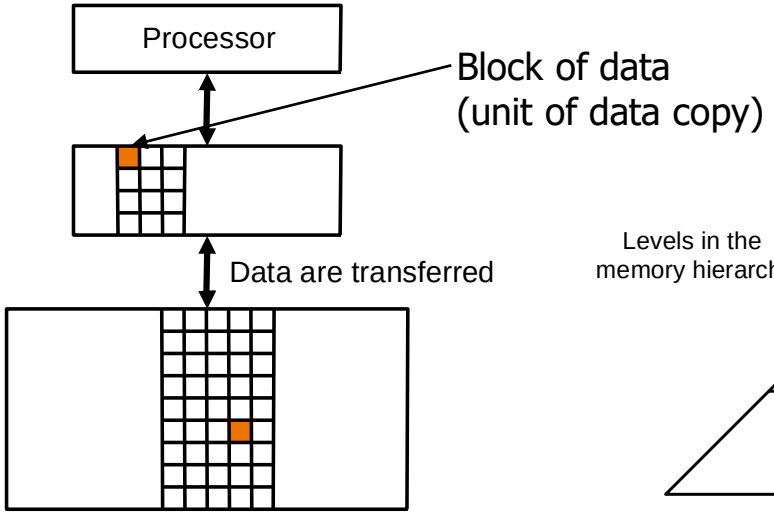
Memories: Review

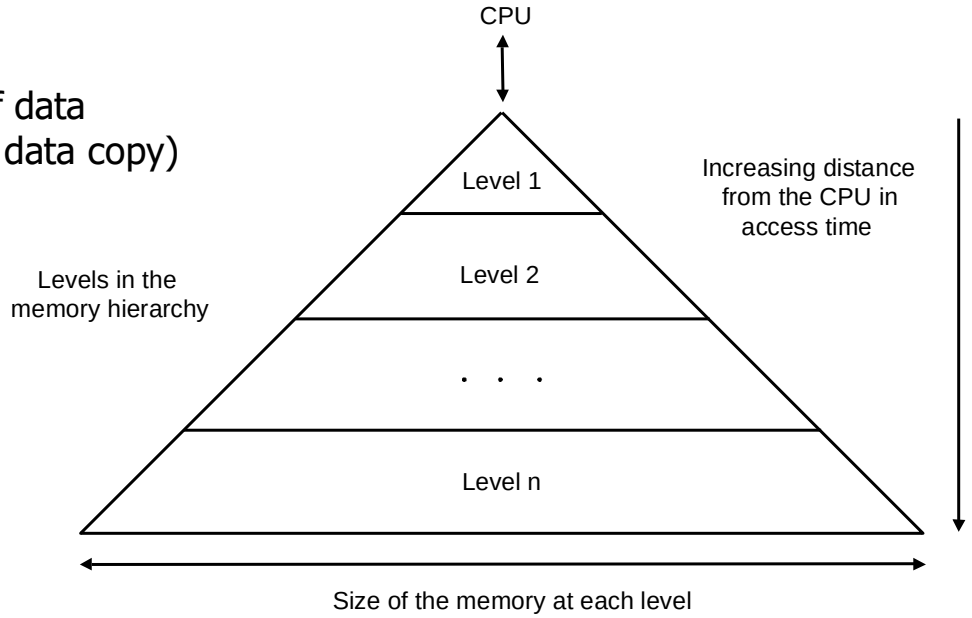
- *DRAM* (Dynamic Random Access Memory):
  - value is stored as a charge on capacitor that must be *periodically refreshed*, which is why it is called *dynamic*
  - very small – 1 transistor per bit – but factor of 5 to 10 slower than SRAM
  - used for *main memory*
- *SRAM* (Static Random Access Memory):
  - value is stored on a pair of inverting gates that will *exist indefinitely* as long as there is power, which is why it is called *static*
  - very fast but takes up more space than DRAM – 4 to 6 transistors per bit
  - used for *cache*

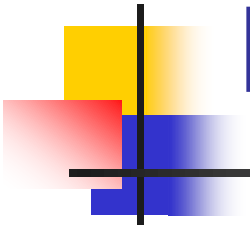


Memory Hierarchy

- Users want large and fast memories…
  - expensive and they don't like to pay…
- Make it seem like they have what they want…
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is *subset* of lower level
  - performance depends on *hit rates*







Locality

- *Locality* is a principle that makes having a memory hierarchy a good idea
- If an item is referenced then because of
  - *temporal locality*: it will tend to be *again* referenced soon
  - *spatial locality*: *nearby items* will tend to be referenced soon
  - *why does code have locality – consider instruction and data?*

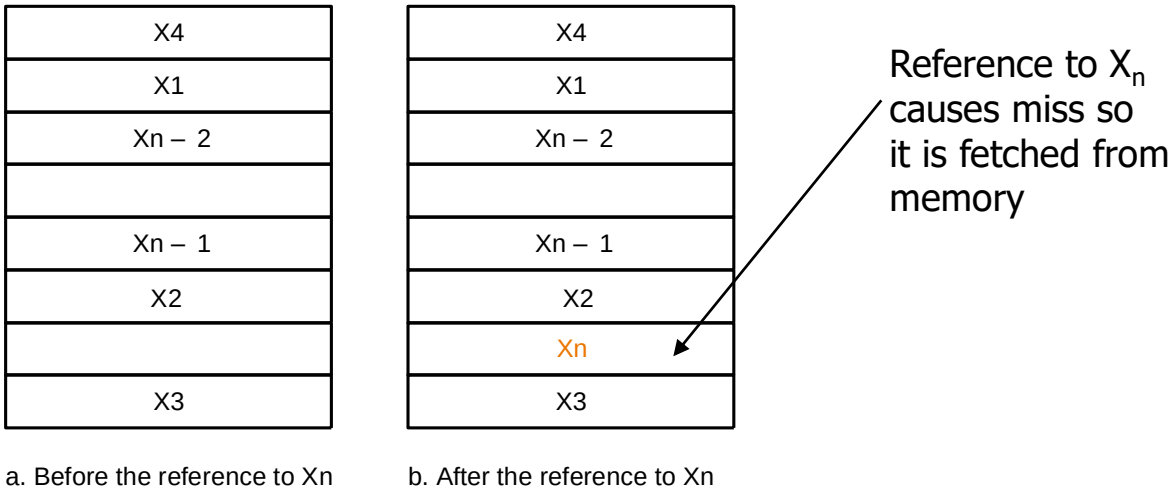
Hit and Miss

- Focus on *any two adjacent* levels – called, *upper* (closer to CPU) and *lower* (farther from CPU) – in the memory hierarchy, because each block copy is always between two adjacent levels
- Terminology:
  - *block*: minimum unit of data to move between levels
  - *hit*: data requested is in upper level
  - *miss*: data requested is not in upper level
  - *hit rate*: fraction of memory accesses that are hits (i.e., found at upper level)
  - *miss rate*: fraction of memory accesses that are not hits
    - miss rate = 1 – hit rate
  - *hit time*: time to determine if the access is indeed a hit + time to access and deliver the data from the upper level to the CPU
  - *miss penalty*: time to determine if the access is a miss + time to replace block at upper level with corresponding block at lower level + time to deliver the block to the CPU

Caches

By simple example

- assume block size = one word of data



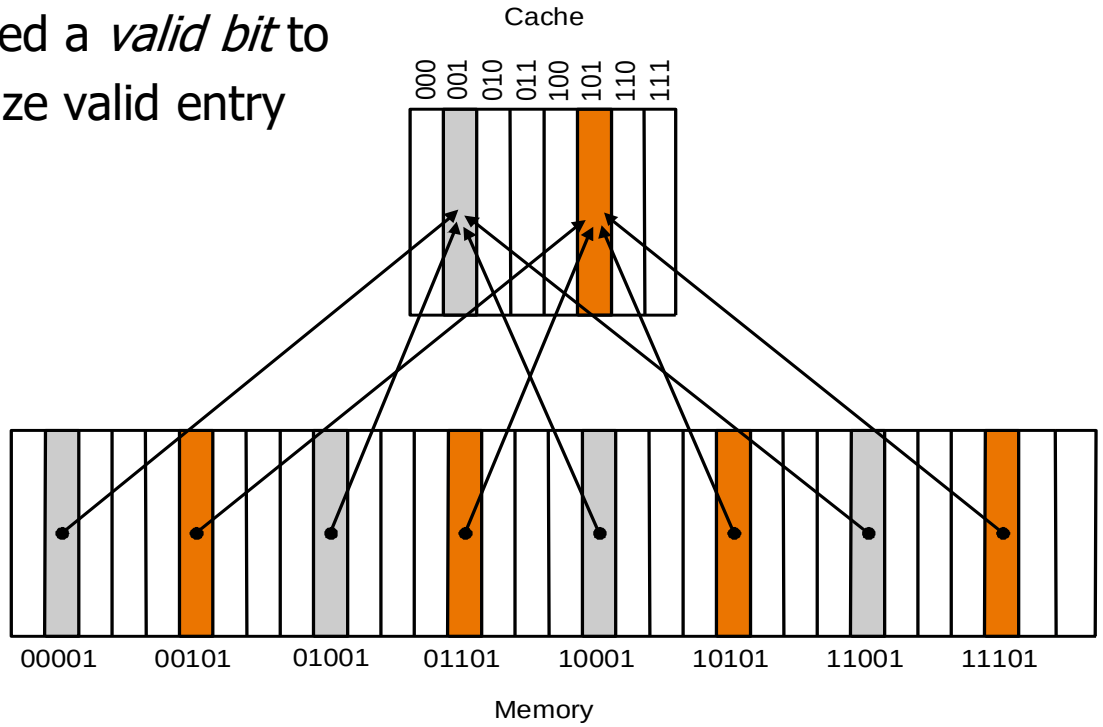
a. Before the reference to Xn

b. After the reference to Xn

- Issues:
  - *how do we know if a data item is in the cache?*
  - *if it is, how do we find it?*
  - *if not, what do we do?*
- Solution depends on *cache addressing scheme*…

Direct Mapped Cache

- Addressing scheme in *direct mapped* cache:
  - cache block address = memory block address *mod* cache size (*unique*)
  - if cache size = 2^m, cache address = lower m bits of n-bit memory address
  - remaining upper n-m bits kept kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry



Accessing Cache

- Example:

(0) Initial state:

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

(1) Address referred 10110 (*miss*):

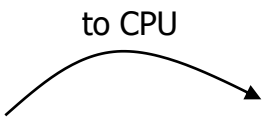
Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(2) Address referred 11010 (*miss*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

(3) Address referred 10110 (*hit*):

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	Mem(11010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

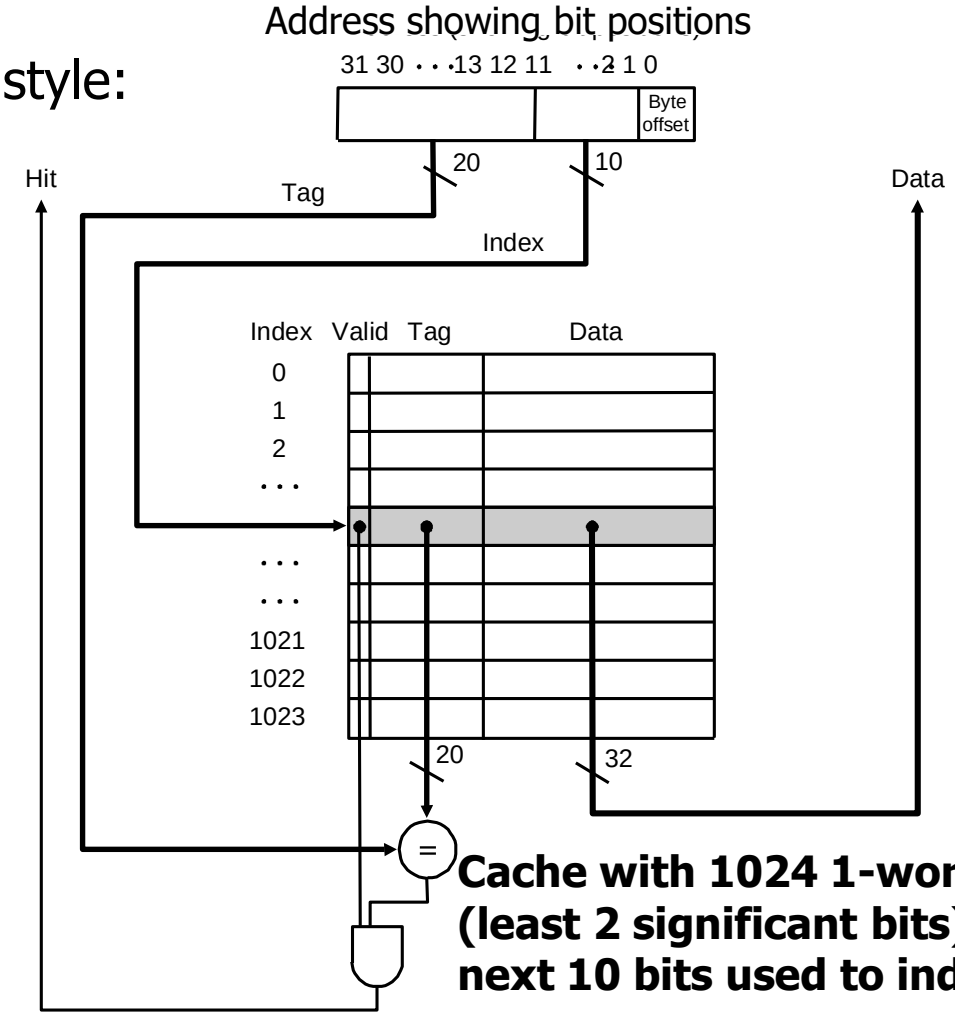


(4) Address referred 10010 (*miss*):

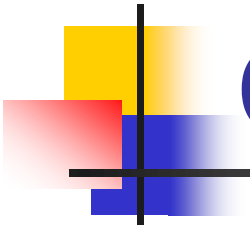
Index	V	Tag	Data
000	N		
001	N		
010	Y	10	Mem(10010)
011	N		
100	N		
101	N		
110	Y	10	Mem(10110)
111	N		

Direct Mapped Cache

- MIPS style:



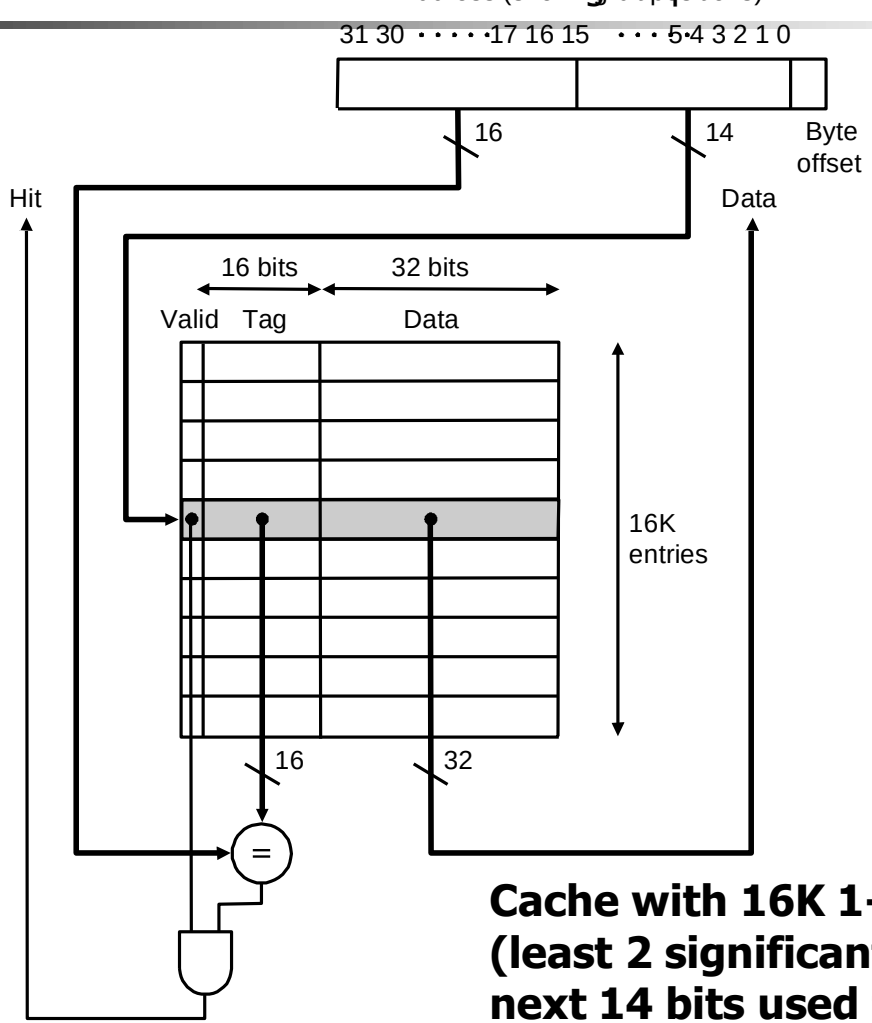
Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache



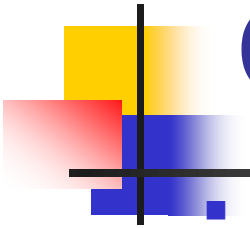
Cache Read Hit/Miss

- *Cache read hit*: no action needed
- *Instruction cache read miss*:
  1. *Send original PC value* (*current PC – 4*, as PC has already been incremented in first step of instruction cycle) to memory
  2. Instruct main memory to perform read and wait for memory to complete access – *stall* on read
  3. After read completes *write cache* entry
  4. *Restart* instruction execution at first step to refetch instruction
- *Data cache read miss*:
  - Similar to instruction cache miss
  - To reduce data miss penalty allow processor to execute instructions while waiting for the read to complete *until* the word is required – *stall on use* (why won't this work for instruction misses?)

DECStation 3100 Cache (MIPS R2000 processor)



Cache with 16K 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 14 bits used to index into cache

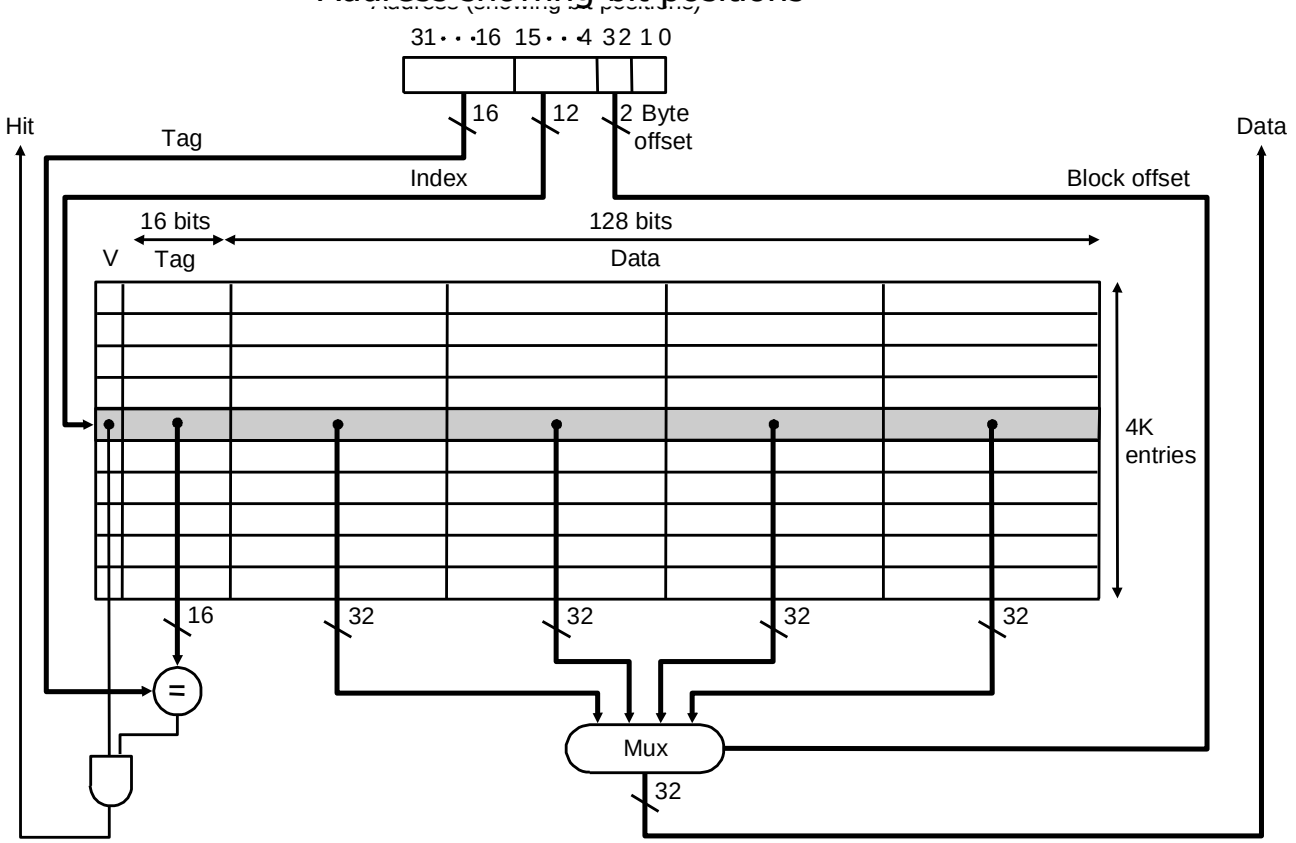


Cache Write Hit/Miss

- *Write-through* scheme
  - on *write hit*: replace data in cache *and* memory with *every* write hit to avoid *inconsistency*
  - on *write miss*: write the word into cache *and* memory – obviously no need to read missed word from memory!
  - Write-through is slow because of always required memory write
    - performance is improved with a *write buffer* where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full
    - when a word in the write buffer completes writing into main that buffer slot is freed and becomes available for future writes
    - DEC 3100 write buffer has 4 words
- *Write-back* scheme
  - write the data block *only* into the cache and *write-back* the block to main *only when* it is replaced in cache
  - more efficient than write-through, more complex to implement

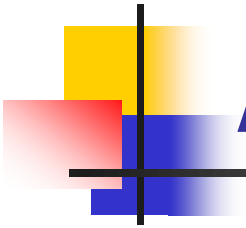
Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:



Cache with 4K 4-word blocks: *byte offset* (least 2 significant bits) is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache

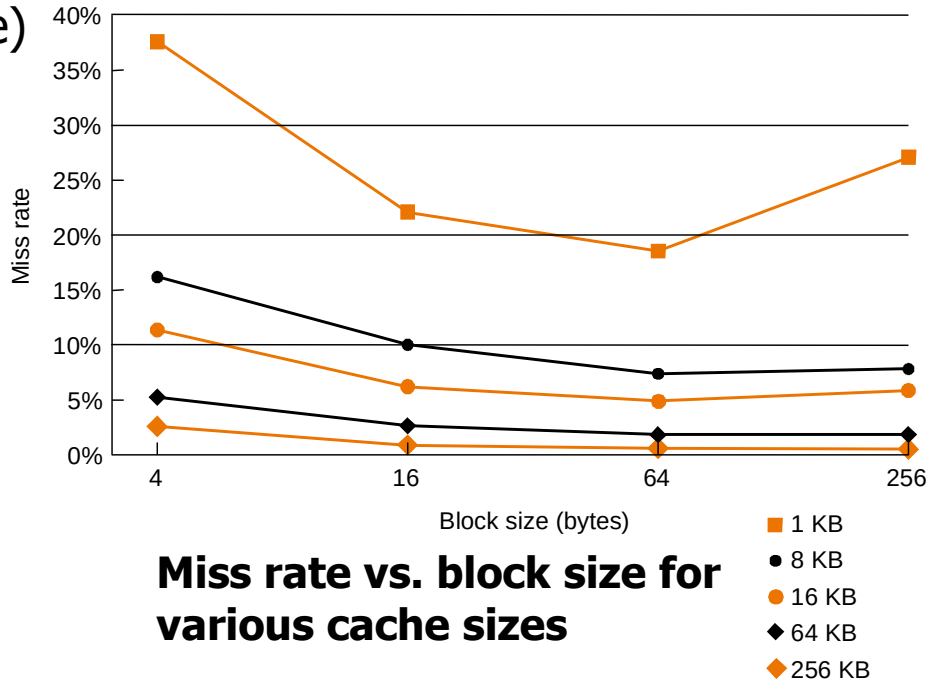
Direct Mapped Cache: Taking Advantage of Spatial Locality



- *Cache replacement* in large (multiword) blocks:
  - word *read miss*: read entire block from main memory
  - word *write miss*: *cannot* simply write word and tag! *Why*?!
  - writing in a *write-through* cache:
    - if *write hit*, i.e., tag of requested address and and cache entry are equal, continue as for 1-word blocks by replacing word and writing block to both cache and memory
    - if *write miss*, i.e., tags are unequal, fetch block from memory, replace word that caused miss, and write block to both cache and memory
    - therefore, unlike case of 1-word blocks, a write miss with a multiword block causes a memory read

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because
  - there are few blocks
  - competition for blocks increases
  - blocks get ejected before most of their words are accessed (*thrashing* in cache)



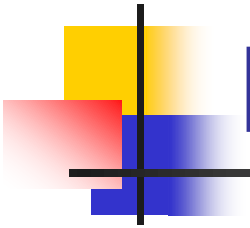
Miss rate vs. block size for various cache sizes

Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 1-word block size, assuming a 32-bit address?*

- Cache data = 128 KB = 2^{17} bytes = 2^{15} words = 2^{15} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 32 + (32 – 15 – 2) + 1 = 48 bits
- Therefore, cache size = $2^{15} \times 48$ bits = $2^{15} \times (1.5 \times 32)$ bits = 1.5×2^{20} bits = 1.5 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.5

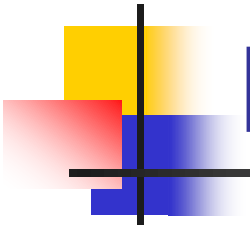


Example Problem

- *How many total bits are required for a direct-mapped cache with 128 KB of data and 4-word block size, assuming a 32-bit address?*

- Cache size = 128 KB = 2^{17} bytes = 2^{15} words = 2^{13} blocks
- Cache entry size = block data bits + tag bits + valid bit

  = 128 + (32 – 13 – 2 – 2) + 1 = 144 bits
- Therefore, cache size = 2^{13} × 144 bits = 2^{13} × (1.25 × 128) bits = 1.25 × 2^{20} bits = 1.25 Mbits
  - data bits in cache = 128 KB × 8 = 1 Mbits
  - total cache size/actual cache data = 1.25



Example Problem

- *Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?*

- As block size = 16 bytes:
  byte address 1200 $\Rightarrow$ block address $\lfloor 1200/16 \rfloor = 75$
- As cache size = 64 blocks:
  block address 75 $\Rightarrow$ cache block (75 *mod* 64) = 11

Improving Cache Performance

- Use split caches for instruction and data because there is more spatial locality in instruction references:

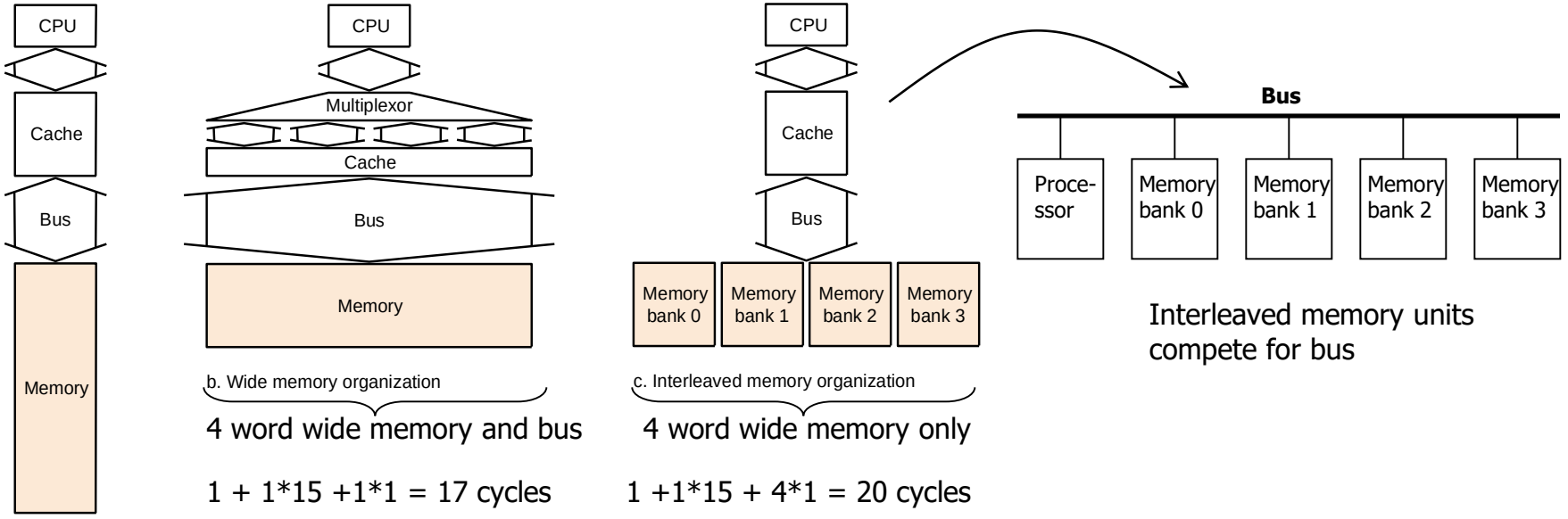
Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Miss rates for gcc and spice in a MIPS R2000 with one and four word block sizes

- Make reading multiple words (higher bandwidth) possible by increasing physical or logical width of the system…

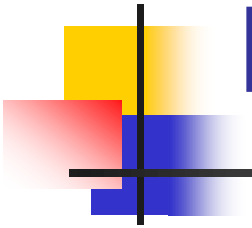
Improving Cache Performance by Increasing Bandwidth

- Assume:
  - cache block of 4 words
  - 1 clock cycle to send address to memory address buffer (1 bus trip)
  - 15 clock cycles for each memory data access
  - 1 clock cycle to send data to memory data buffer (1 bus trip)



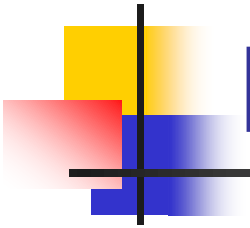
1 + 4*15 + 4*1 = 65 cycles

Miss penalties



Performance

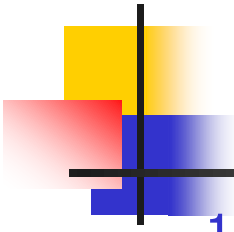
- Simplified model assuming equal read and write miss penalties:
  - CPU time = (execution cycles + memory stall cycles) × cycle time
  - memory stall cycles = memory accesses × miss rate × miss penalty
- Therefore, two ways to improve performance in cache:
  - decrease miss rate
  - decrease miss penalty
  - *what happens if we increase block size?*



Example Problems

- Assume for a given machine and program:
  - instruction cache miss rate 2%
  - data cache miss rate 4%
  - miss penalty always 40 cycles
  - CPI of 2 without memory stalls
  - frequency of load/stores 36% of instructions

1. *How much faster is a machine with a perfect cache that never misses?*
2. *What happens if we speed up the machine by reducing its CPI to 1 without changing the clock rate?*
3. *What happens if we speed up the machine by doubling its clock rate, but if the absolute time for a miss penalty remains same?*



Solution

1.

- Assume instruction count = I
- Instruction miss cycles = I × 2% × 40 = 0.8 × I
- Data miss cycles = I × 36% × 4% × 40 = 0.576 × I
- So, total memory-stall cycles = 0.8 × I + 0.576 × I = 1.376 × I
  - in other words, 1.376 stall cycles per instruction
- Therefore, CPI with memory stalls = 2 + 1.376 = 3.376
- Assuming instruction count and clock rate remain same for a perfect cache and a cache that misses:

  CPU time with stalls / CPU time with perfect cache

  = 3.376 / 2 = 1.688
- Performance with a perfect cache is better by a factor of 1.688

Solution (cont.)

2.

- CPI without stall = 1
- CPI with stall = 1 + 1.376 = 2.376 (clock has not changed so stall cycles per instruction remains same)
- CPU time with stalls / CPU time with perfect cache

  = CPI with stall / CPI without stall

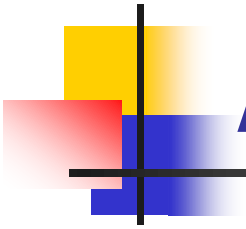
  = 2.376
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher CPI cache misses “hurt more” than with lower CPI

Solution (cont.)

3.

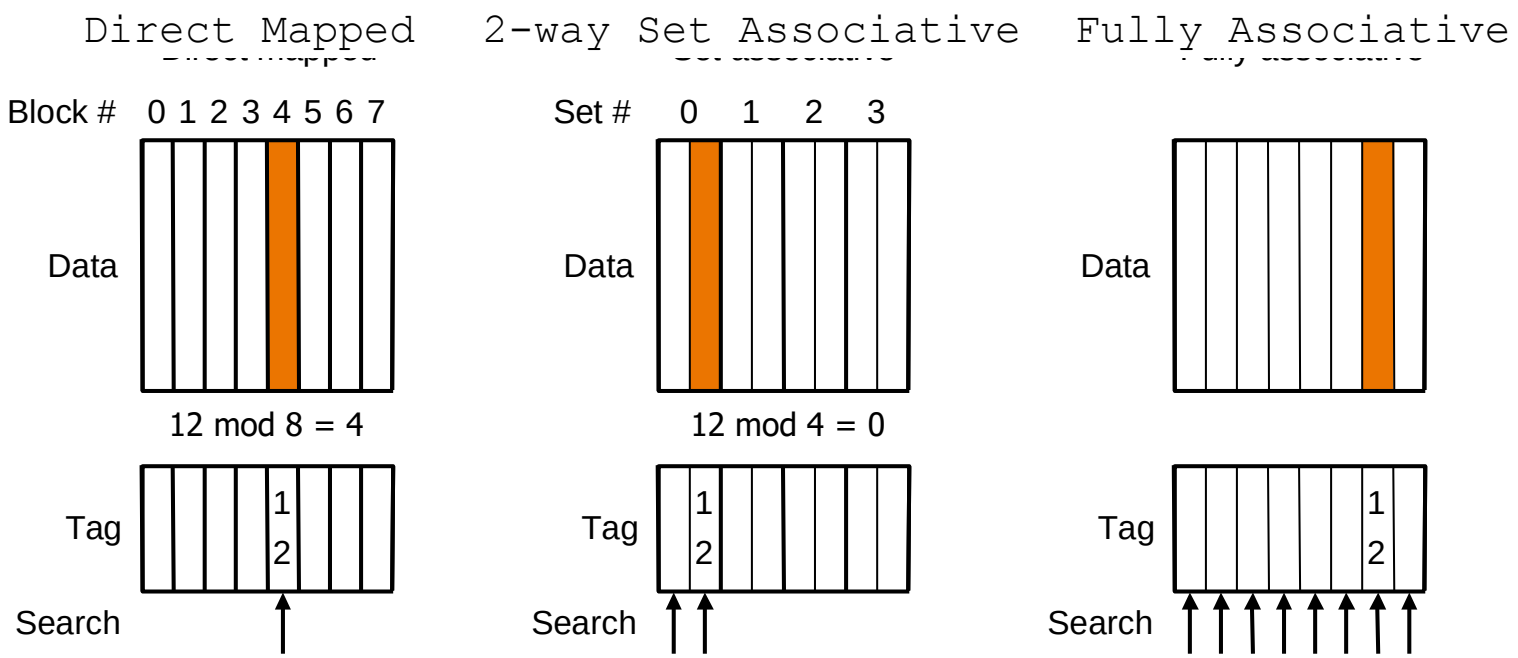
- With doubled clock rate, miss penalty = 2 × 40 = 80 clock cycles
- Stall cycles per instruction = (I × 2% × 80) + (I × 36% × 4% × 80)
  = 2.752 × I
- So, faster machine with cache miss has CPI = 2 + 2.752 = 4.752
- CPU time with stalls / CPU time with perfect cache
  = CPI with stall / CPI without stall
  = 4.752 / 2 = 2.376
- Performance with a perfect cache is better by a factor of 2.376
- Conclusion: with higher clock rate cache misses “hurt more” than with lower clock rate

Decreasing Miss Rates with Associative Block Placment



- *Direct mapped*: one *unique* cache location for each memory block
  - cache block address = memory block address *mod* cache size
- *Fully associative*: each memory block can locate *anywhere* in cache
  - *all* cache entries are searched (in parallel) to locate block
- *Set associative*: each memory block can place in a *unique set* of cache locations – if the set is of size n it is n-way set-associative
  - cache set address = memory block address *mod* number of sets in cache
  - all cache entries in the corresponding set are searched (*in parallel*) to locate block
- Increasing degree of associativity
  - *reduces miss rate*
  - *increases hit time* because of the parallel search and then fetch

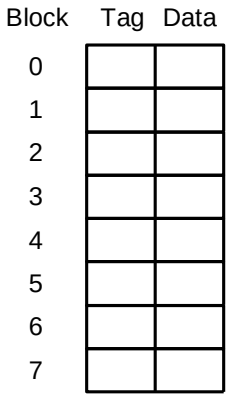
Decreasing Miss Rates with Associative Block Placment



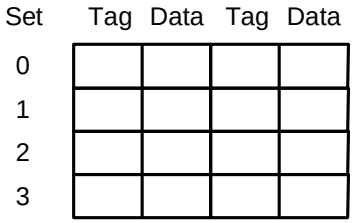
Location of a memory block with address 12 in a cache with 8 blocks with different degrees of associativity

Decreasing Miss Rates with Associative Block Placment

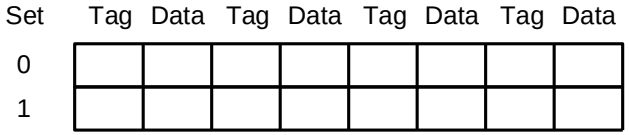
One-way set associative
(direct mapped)



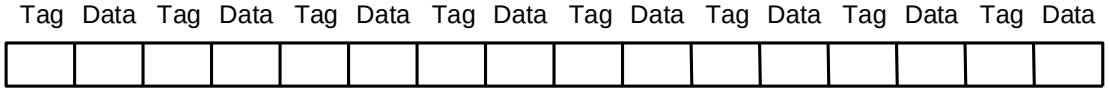
Two-way set associative



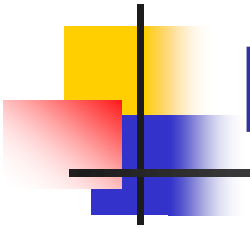
Four-way set associative



Eight-way set associative (fully associative)



Configurations of an 8-block cache with different degrees of associativity



Example Problems

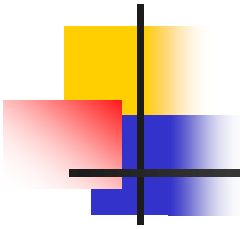
- *Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:*

  0, 8, 0, 6, 8,

  for each of the following cache configurations

  1. *direct mapped*
  2. *2-way set associative (use LRU replacement policy)*
  3. *fully associative*

- Note about LRU replacement
  - in a 2-way set associative cache LRU replacement can be implemented with one bit at each set whose value indicates the mostly recently referenced block



Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 (= 0 *mod* 4)
6	2 (= 6 *mod* 4)
8	0 (= 8 *mod* 4)

Block address translation in direct-mapped cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference 0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Cache contents after each reference – red indicates new entry added

- 5 misses

Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 (= 0 *mod* 2)
6	0 (= 6 *mod* 2)
8	0 (= 8 *mod* 2)

Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Cache contents after each reference – red indicates new entry added

- Four misses

Solution (cont.)

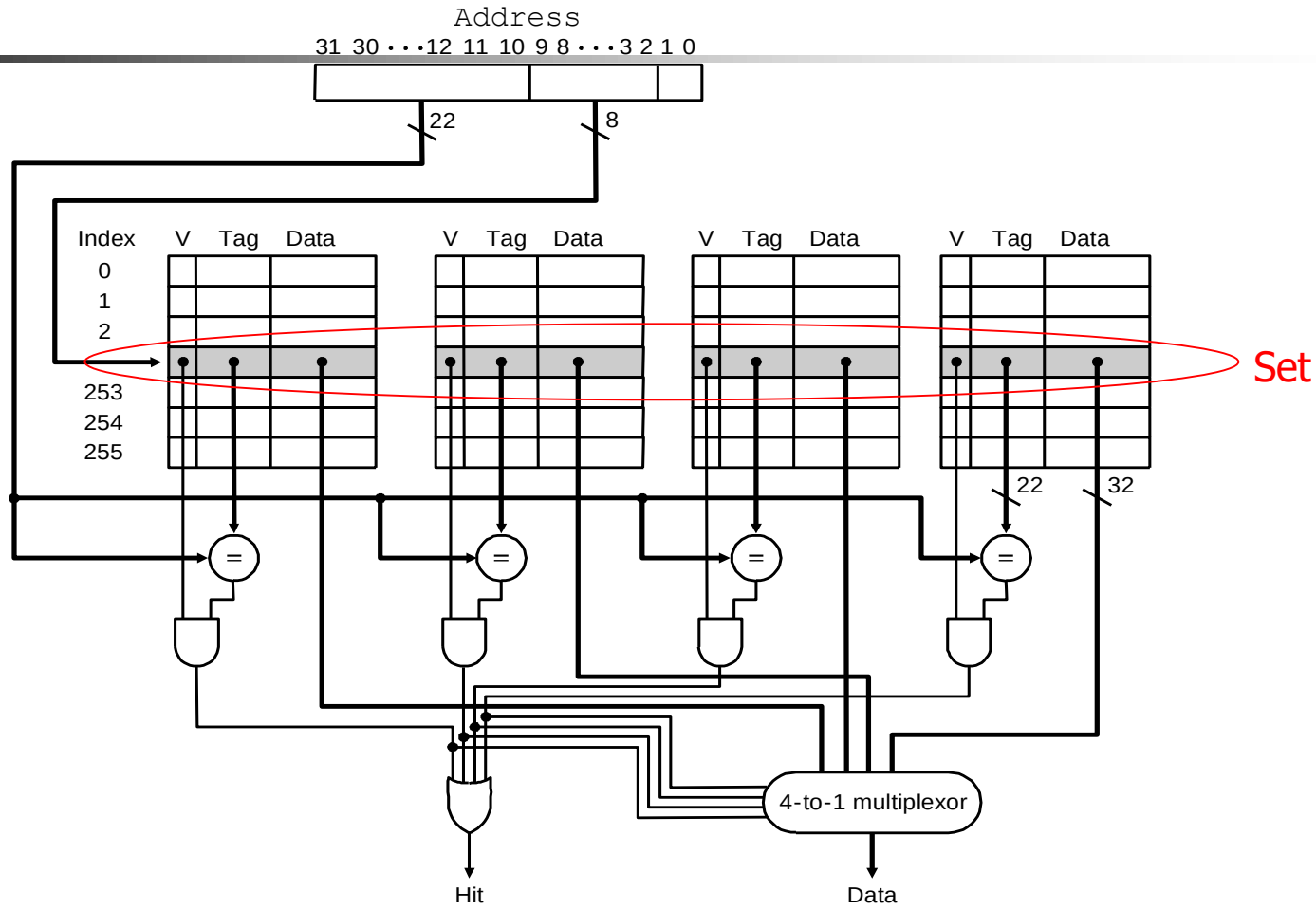
- 3 (fully associative)

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

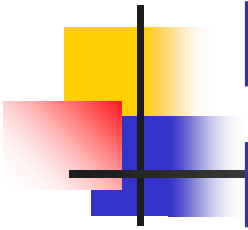
Cache contents after each reference – red indicates new entry added

- 3 misses

Implementation of a Set-Associative Cache



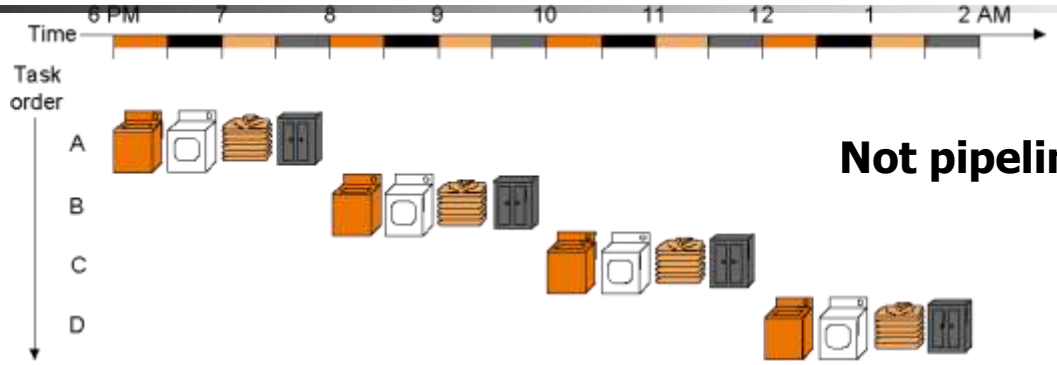
4-way set-associative cache with 4 comparators and one 4-to-1 multiplexor: size of cache is 1K blocks = 256 sets * 4-block set size



COD Ch. 6 Enhancing Performance with Pipelining

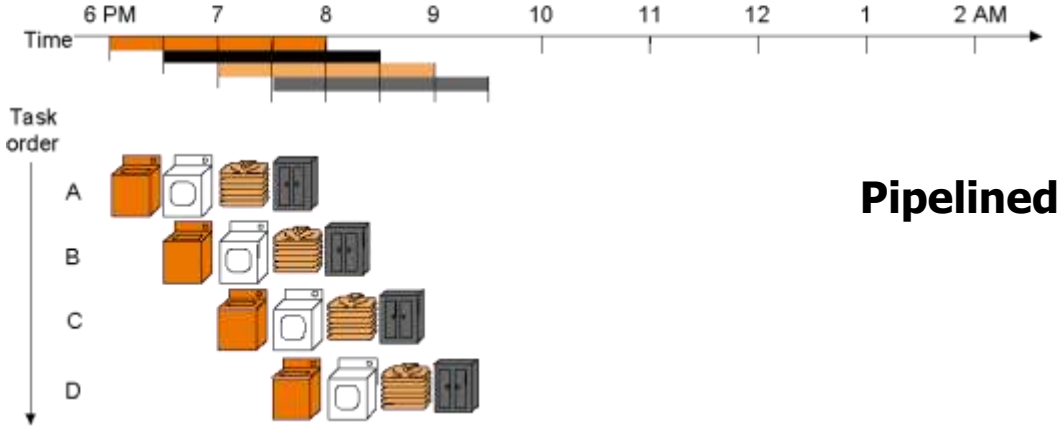
Pipelining

- Start work ASAP!! Do not waste time!



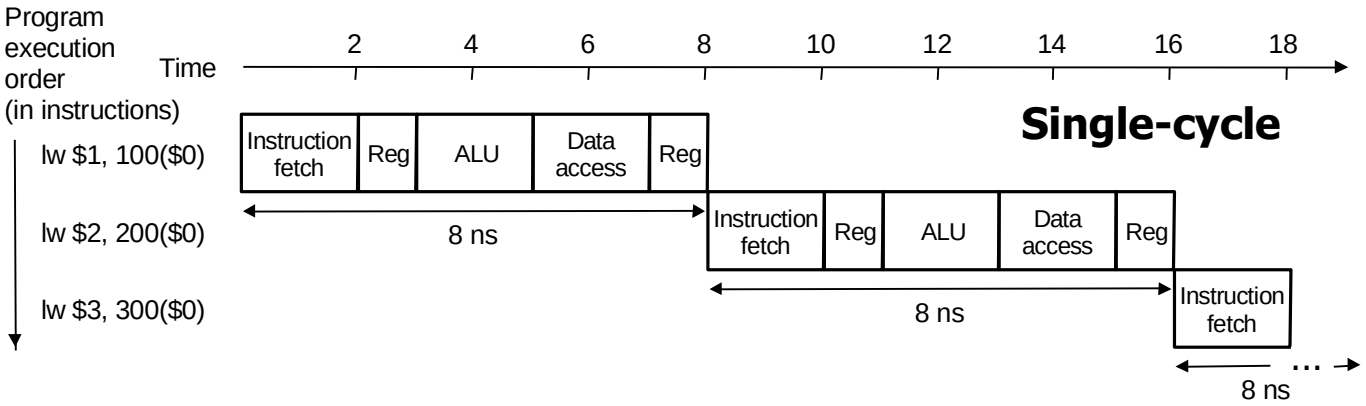
Not pipelined

Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

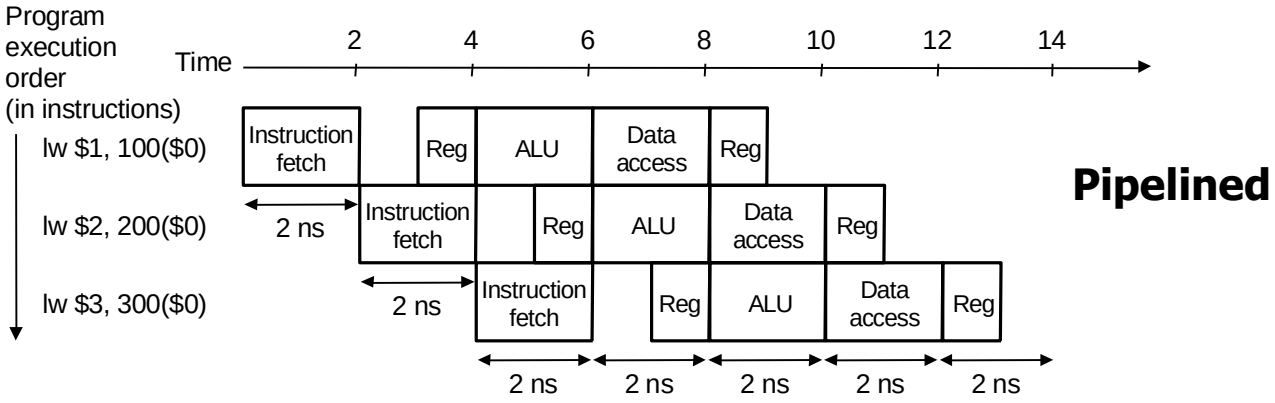


Pipelined

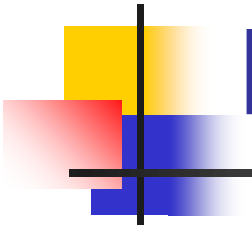
Pipelined vs. Single-Cycle Instruction Execution: the Plan



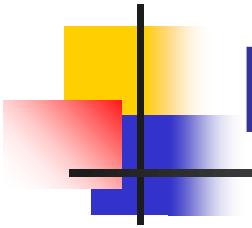
Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Pipelining: Keep in Mind



- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

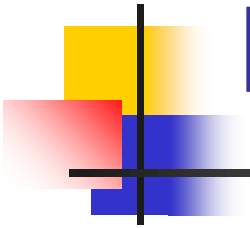


Example Problem

- *Problem: for the laundry fill in the following table when*
  1. *the stage lengths are 30, 30, 30 30 min., resp.*
  2. *the stage lengths are 20, 20, 60, 20 min., resp.*

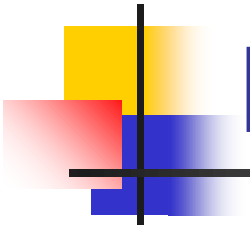
Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1	Pipeline 2 finish time	Ratio unpiplelined to pipeline 2
1					
2					
3					
4					
n					

- *Come up with a formula for pipeline speed-up!*



Pipelining MIPS

- What makes it easy with MIPS?
  - all *instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

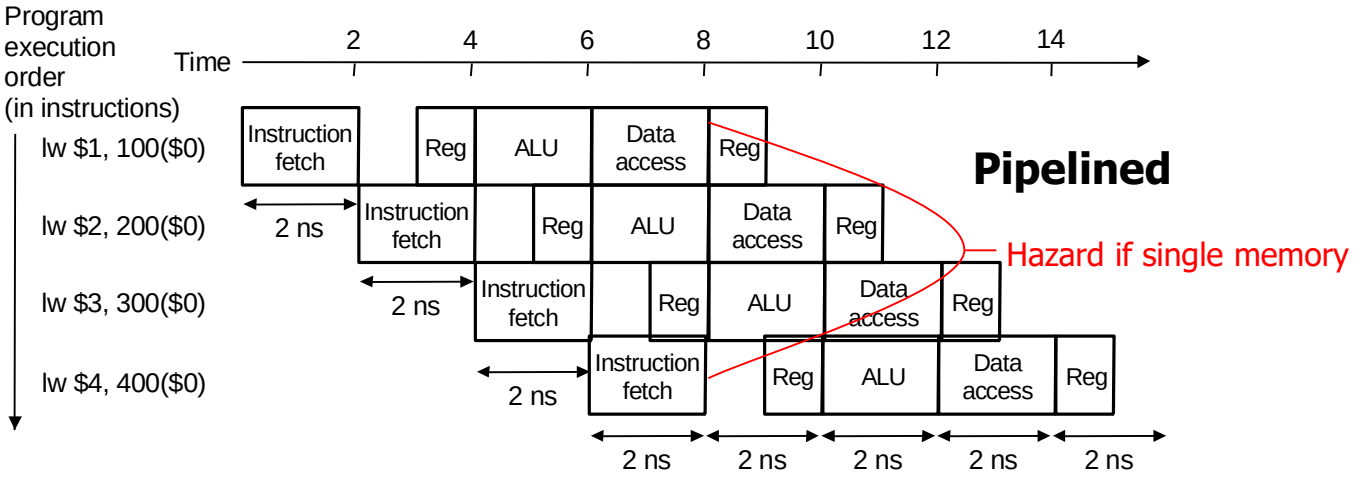


Pipelining MIPS

- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

Structural Hazards

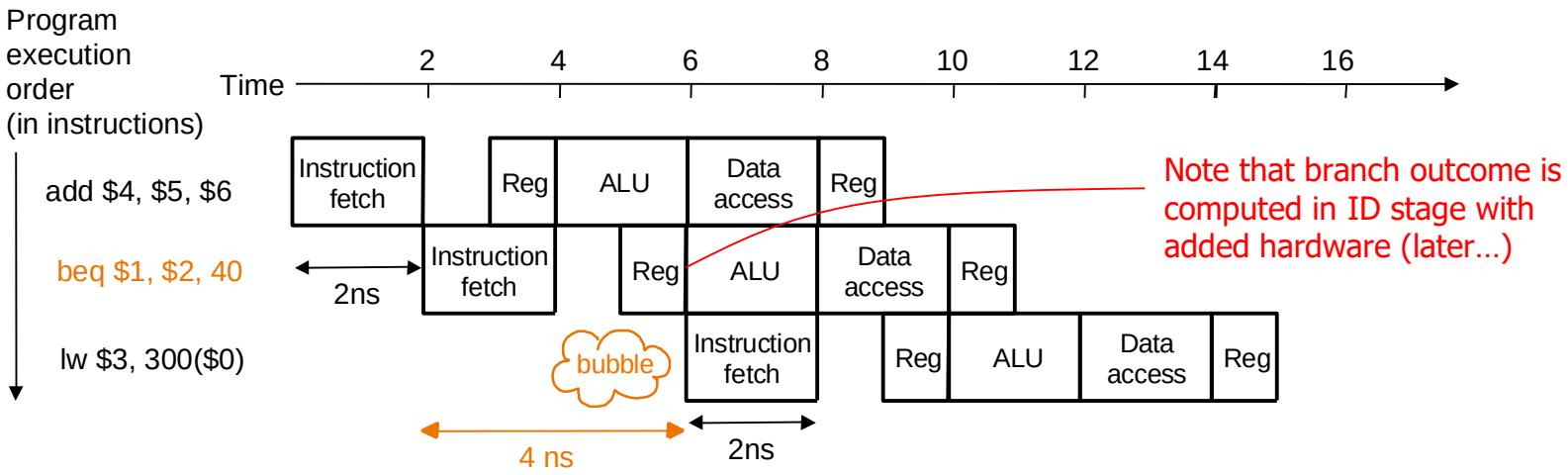
- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth `lw` instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

Control Hazards

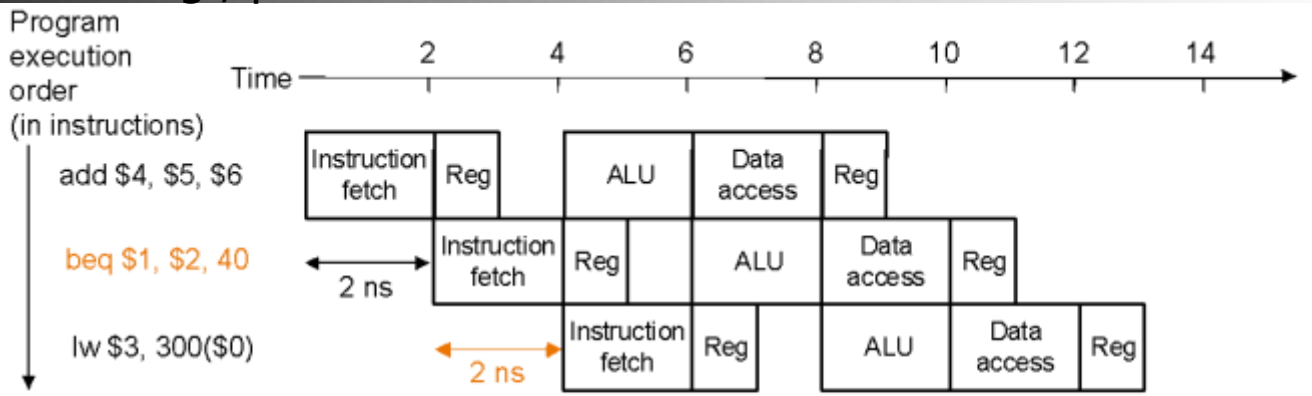
- *Control hazard*: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1 *Stall* the pipeline



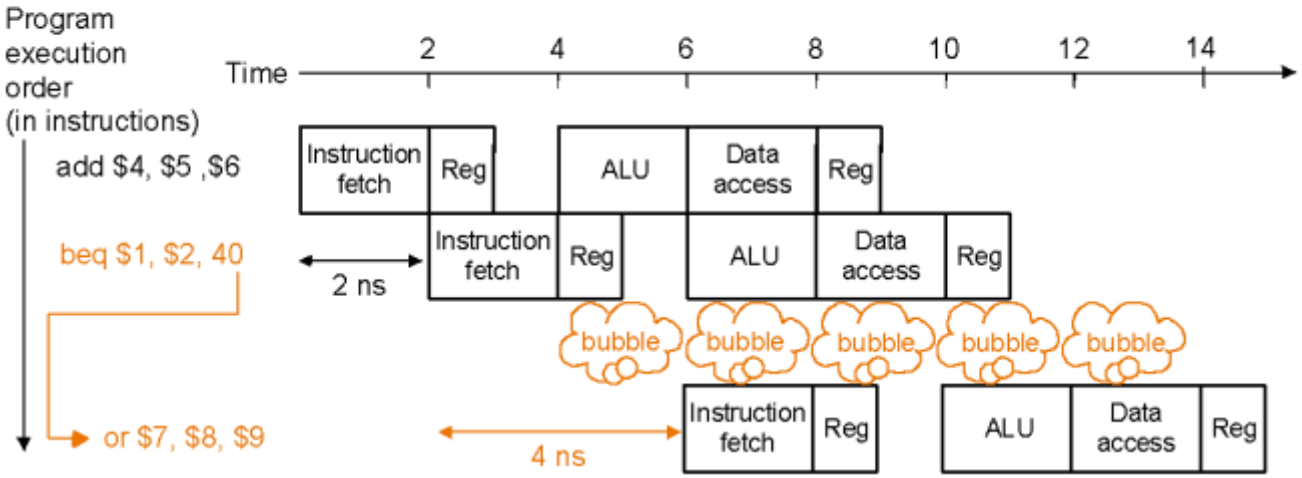
Pipeline stall

Control Hazards

- Solution 2 *Predict* branch outcome
  - e.g., predict *branch-not-taken* :



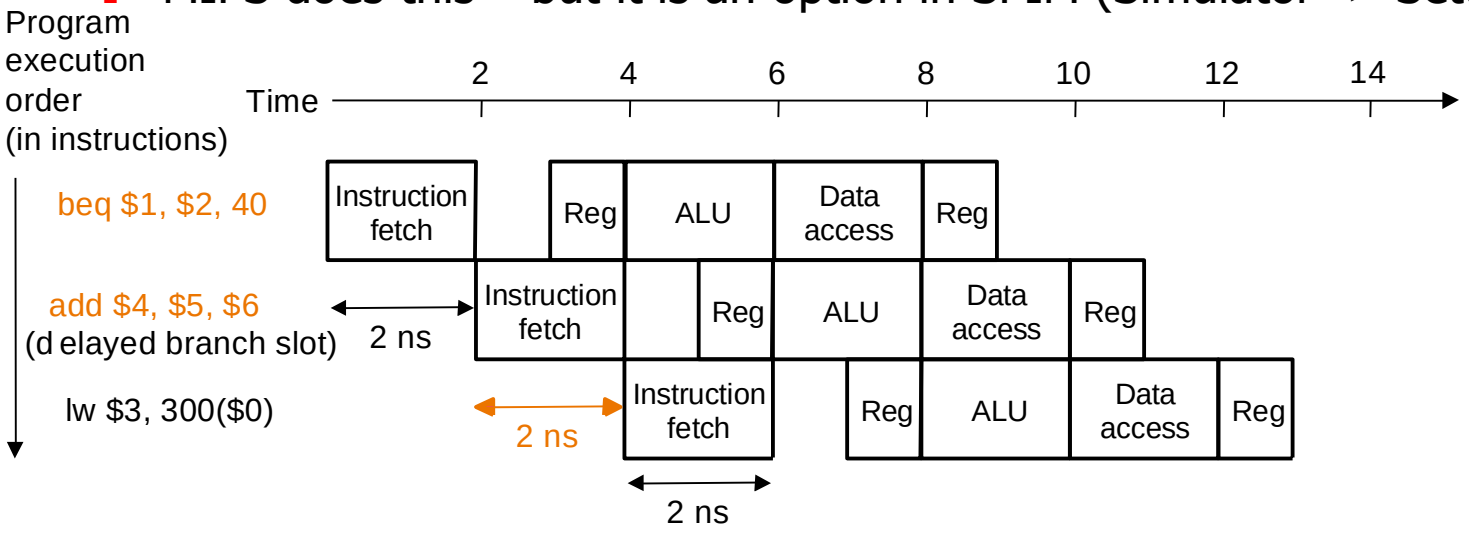
Prediction success



Prediction failure: undo (=flush) `lw`

Control Hazards

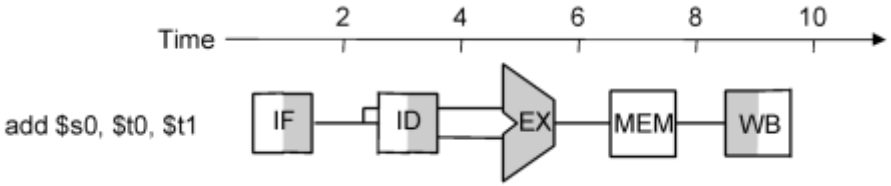
- Solution 3 *Delayed branch:* always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



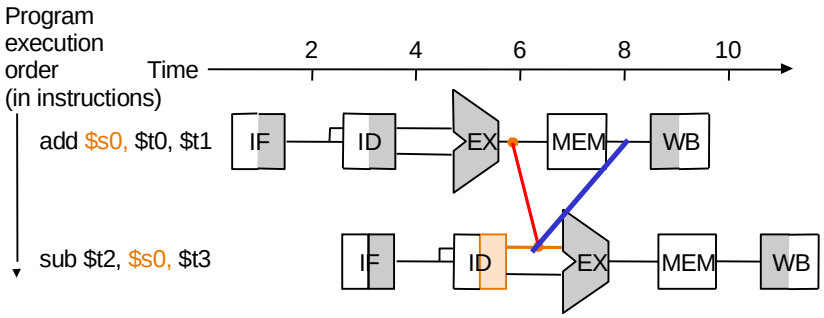
Delayed branch `beq` is followed by `add` that is independent of branch outcome

Data Hazards

- *Data hazard*: instruction needs data from the result of a previous instruction still executing in pipeline
- Solution *Forward* data if possible…



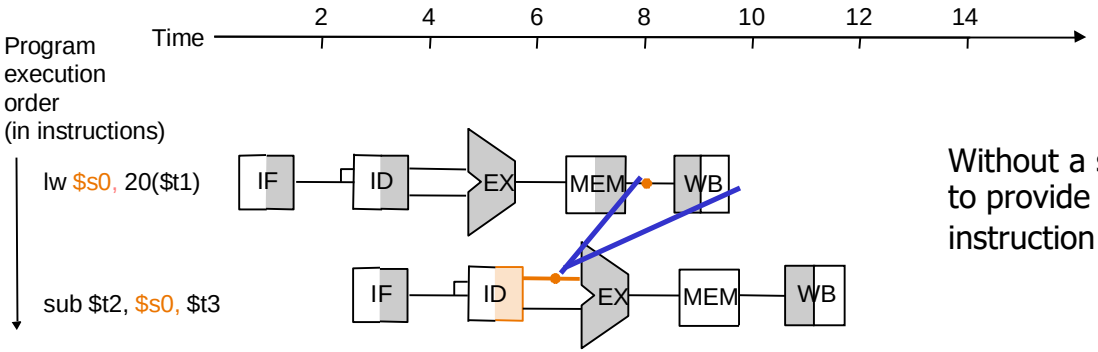
Instruction pipeline diagram: shade indicates use – left=write, right=read



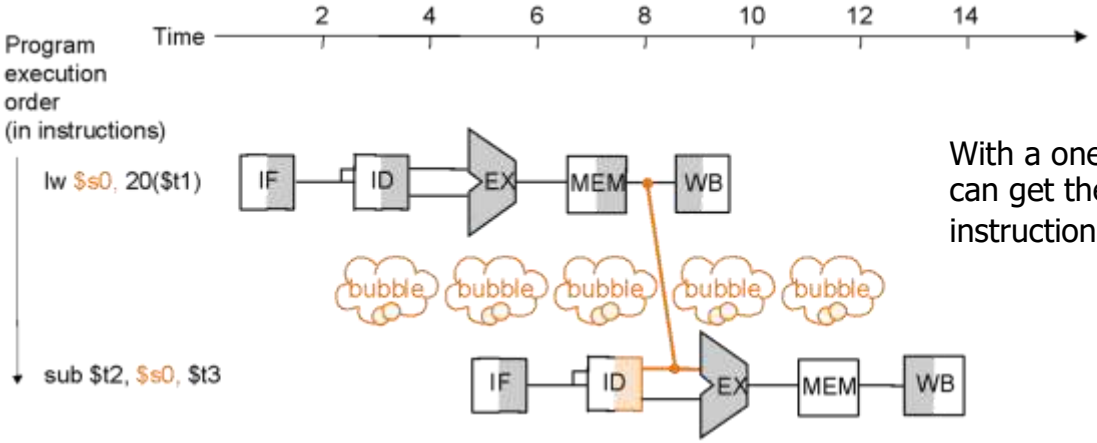
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*



Without a stall it is impossible to provide input to the `sub` instruction in time



With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

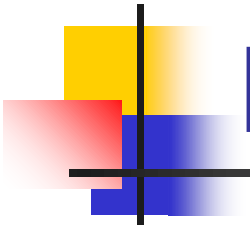
```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

- Reordered code:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

Interchanged

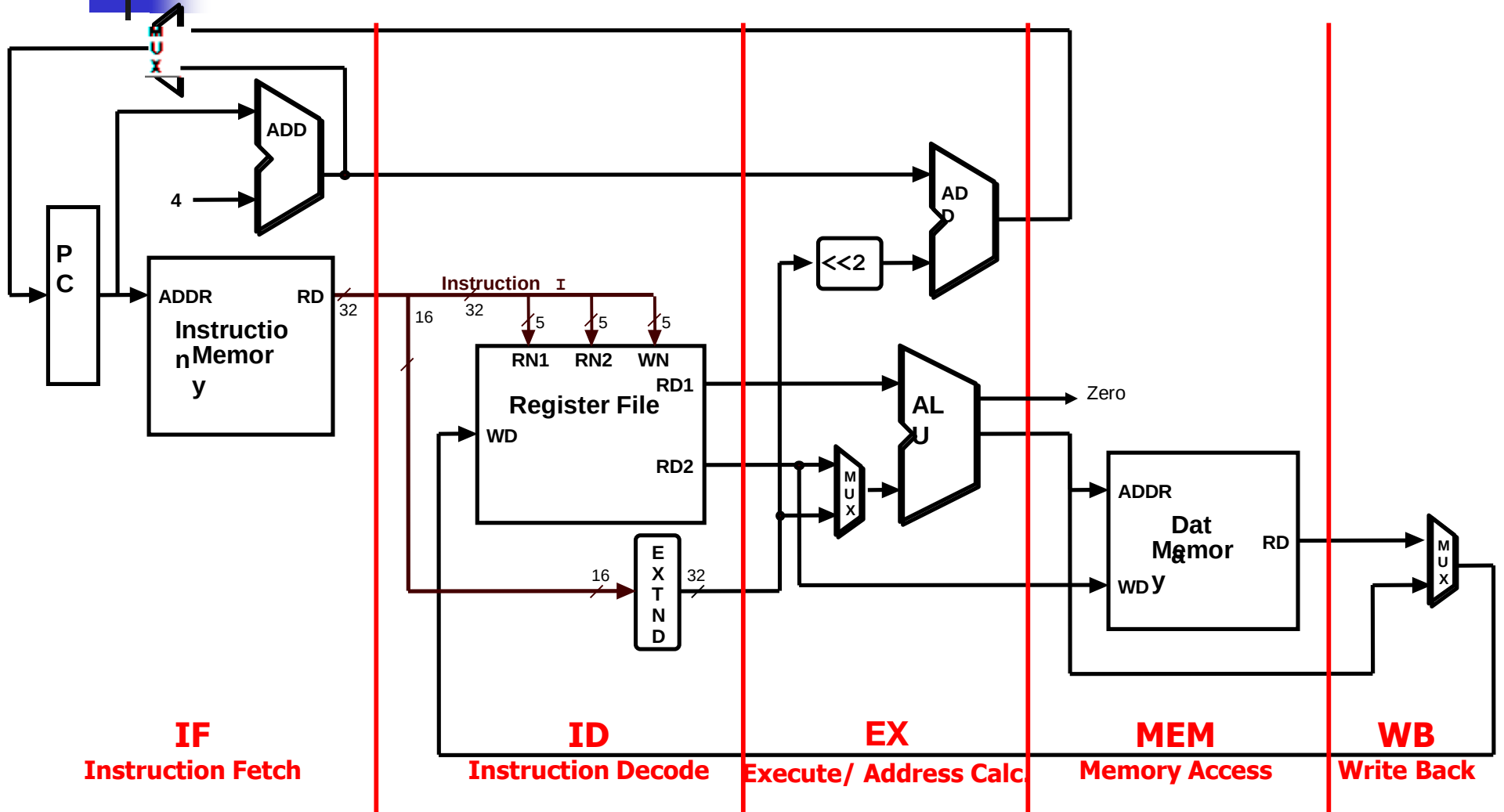


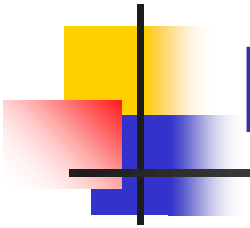
Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

Review - Single-Cycle Datapath "Steps"

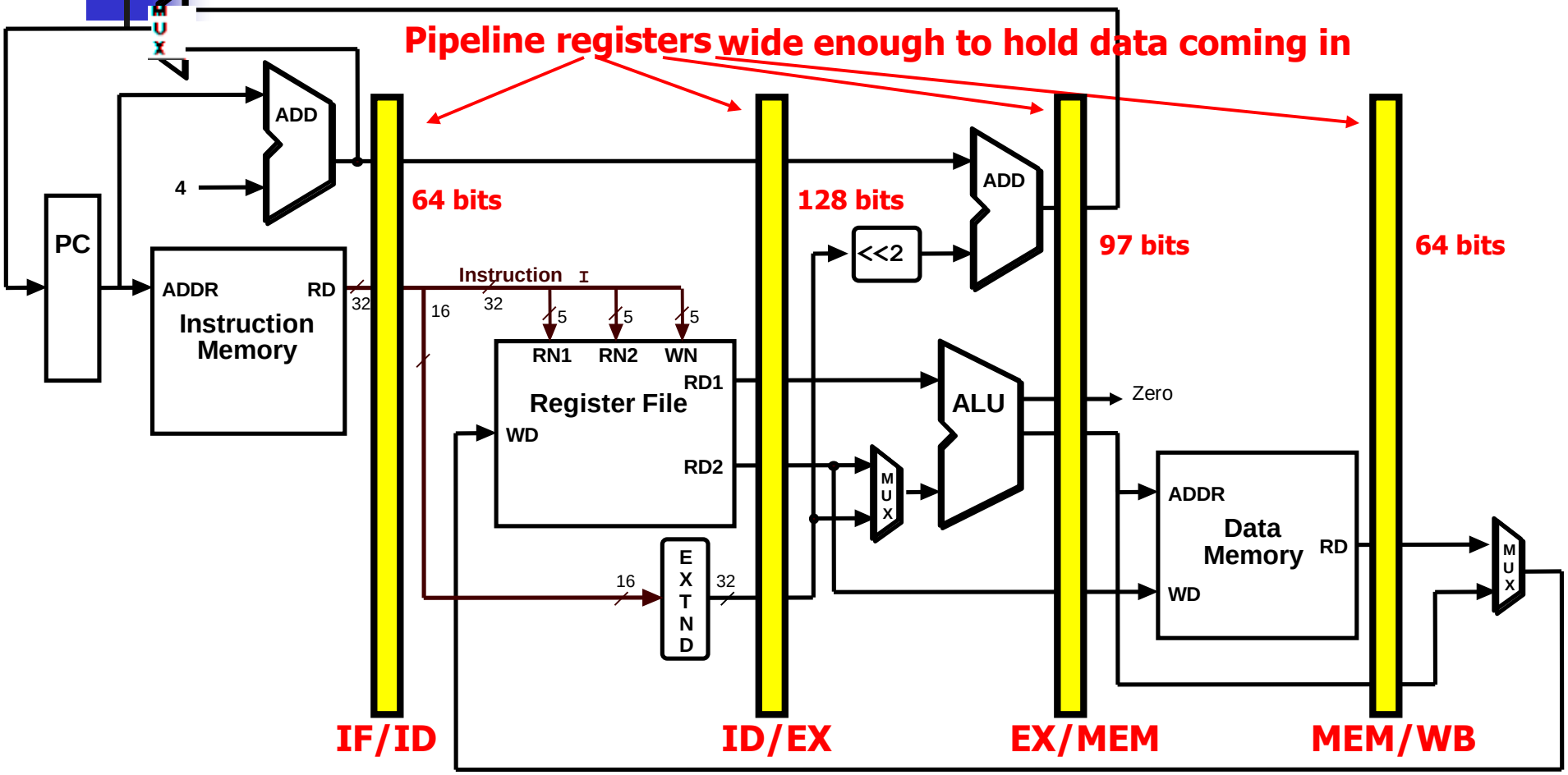


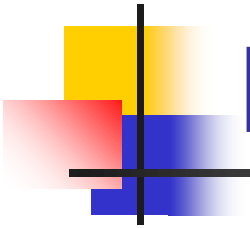


Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - Answer: *We may be able to start executing a new instruction at each clock cycle* - pipelining
- …but we shall need *extra* registers to hold data between cycles – *pipeline registers*

Pipelined Datapath



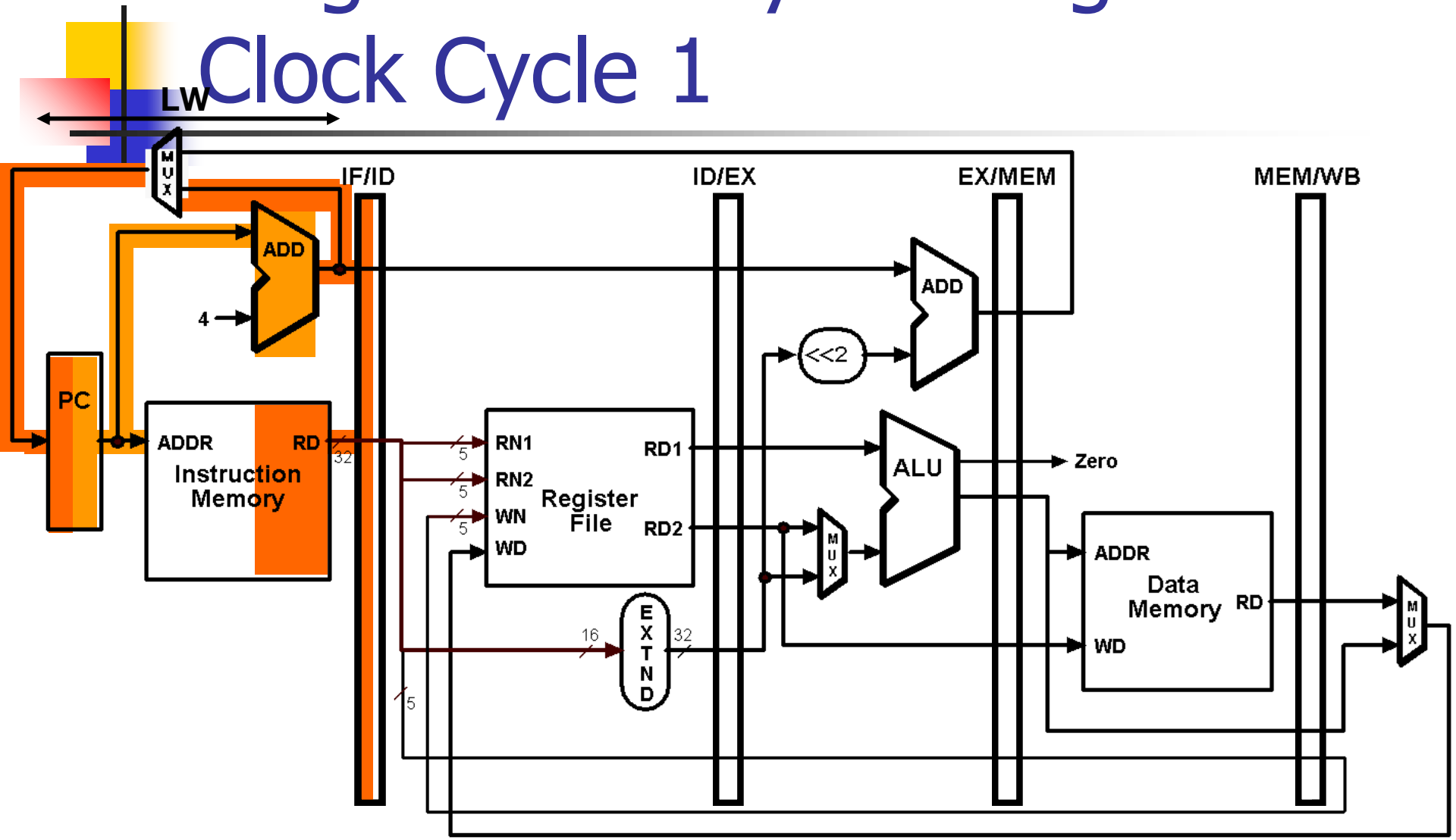


Pipelined Example

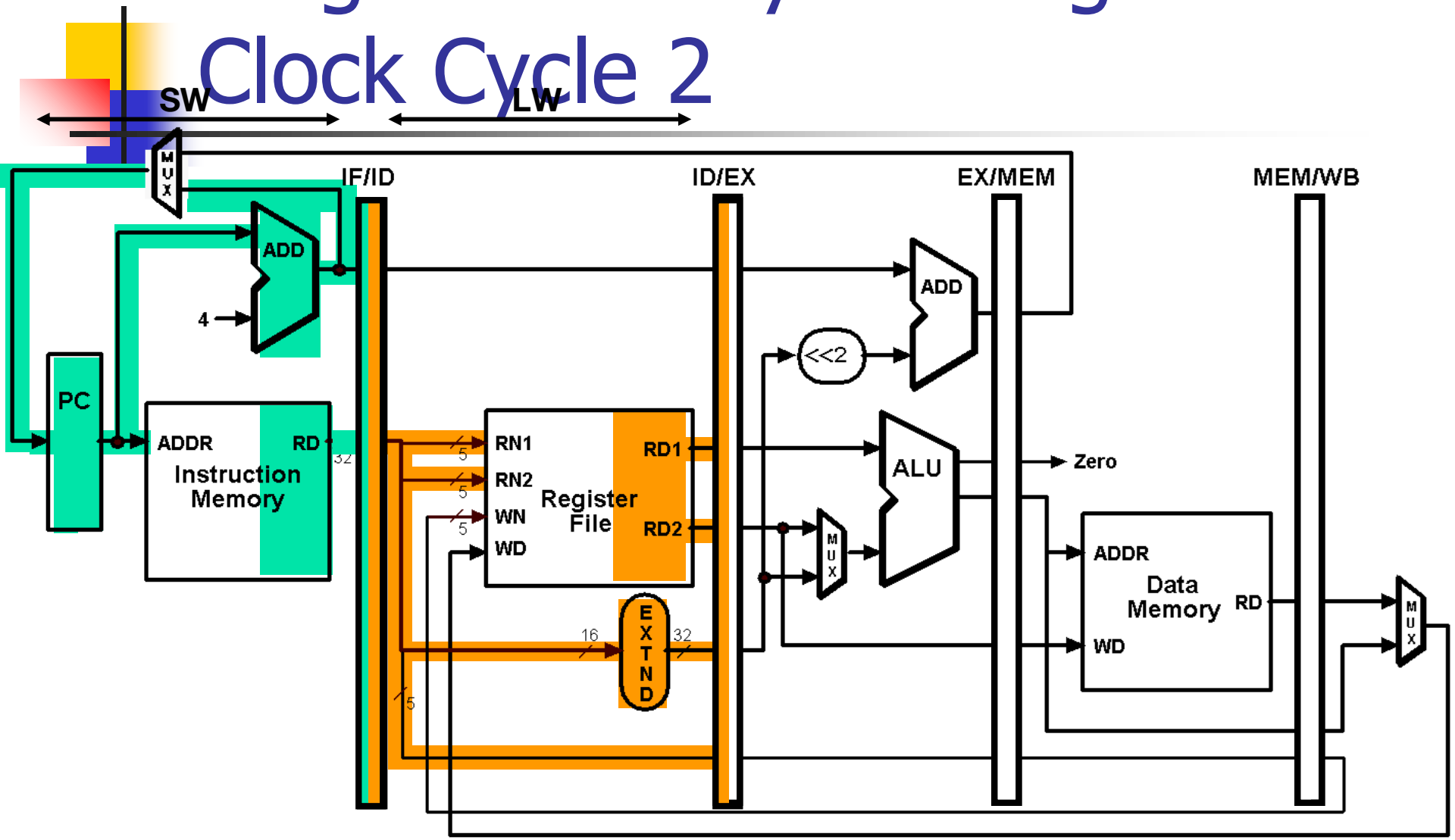
- Consider the following instruction sequence:

```
lw   $t0,  10($t1)
sw   $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

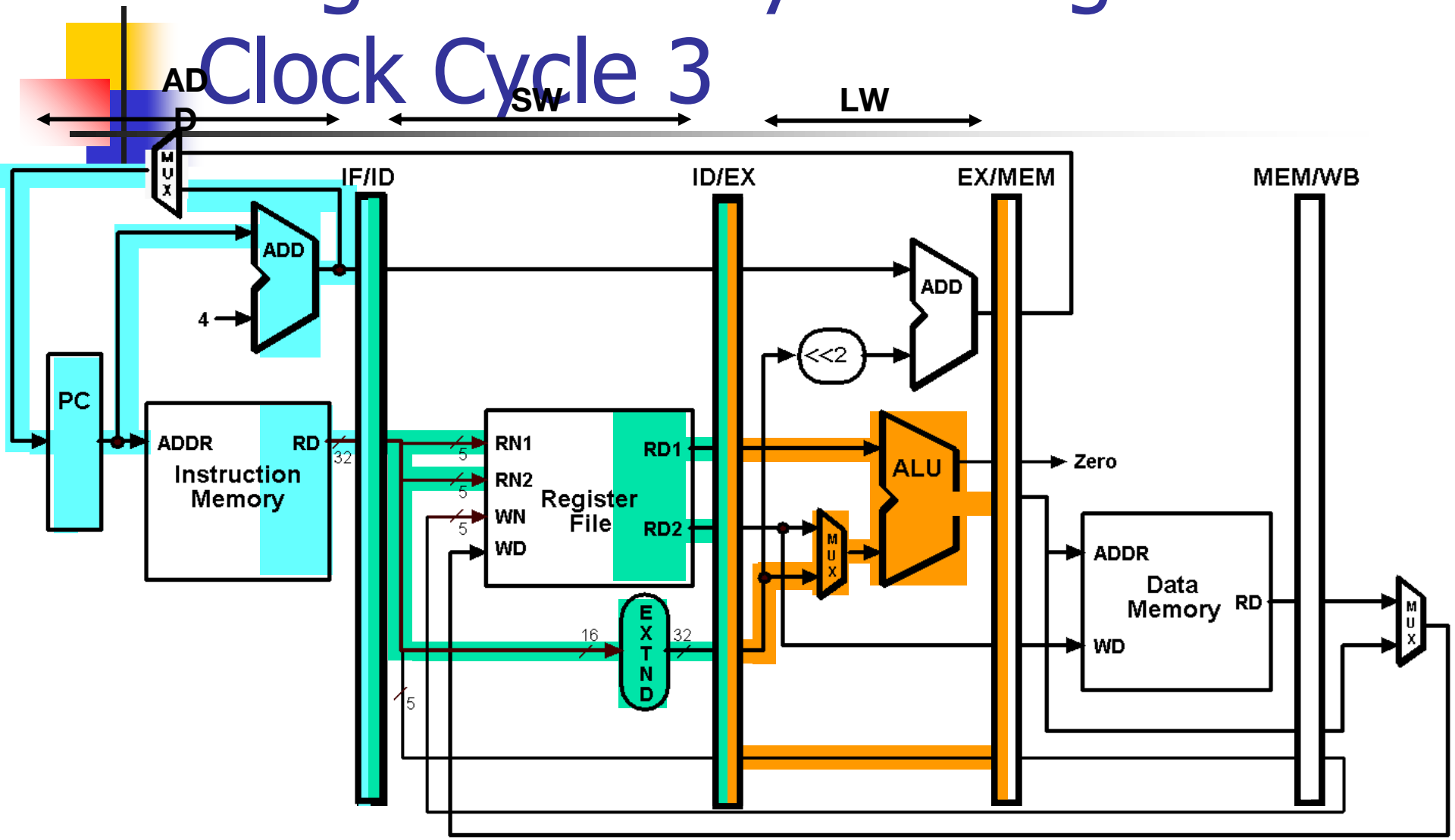
Single-Clock-Cycle Diagram: Clock Cycle 1



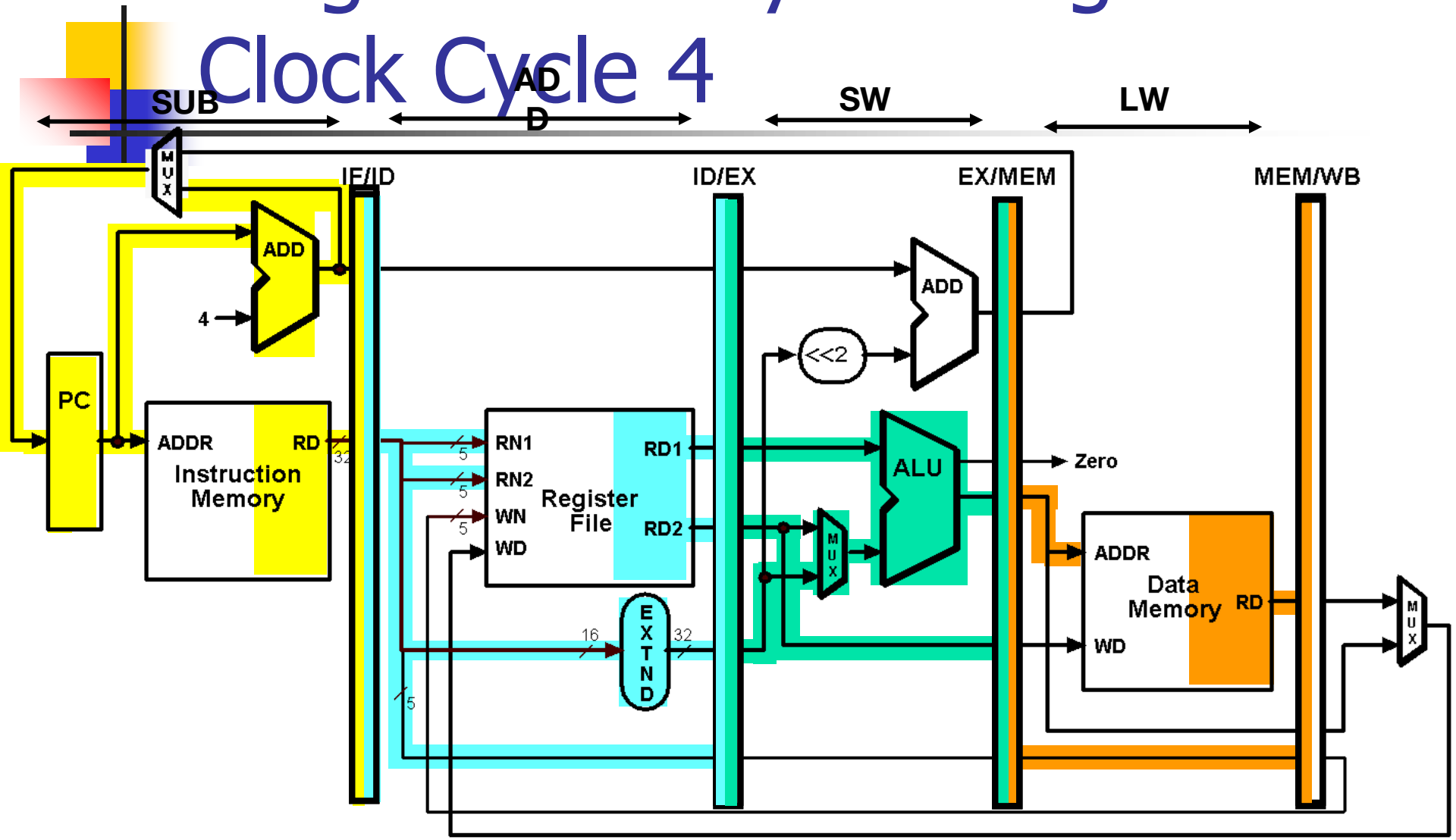
Single-Clock-Cycle Diagram: Clock Cycle 2



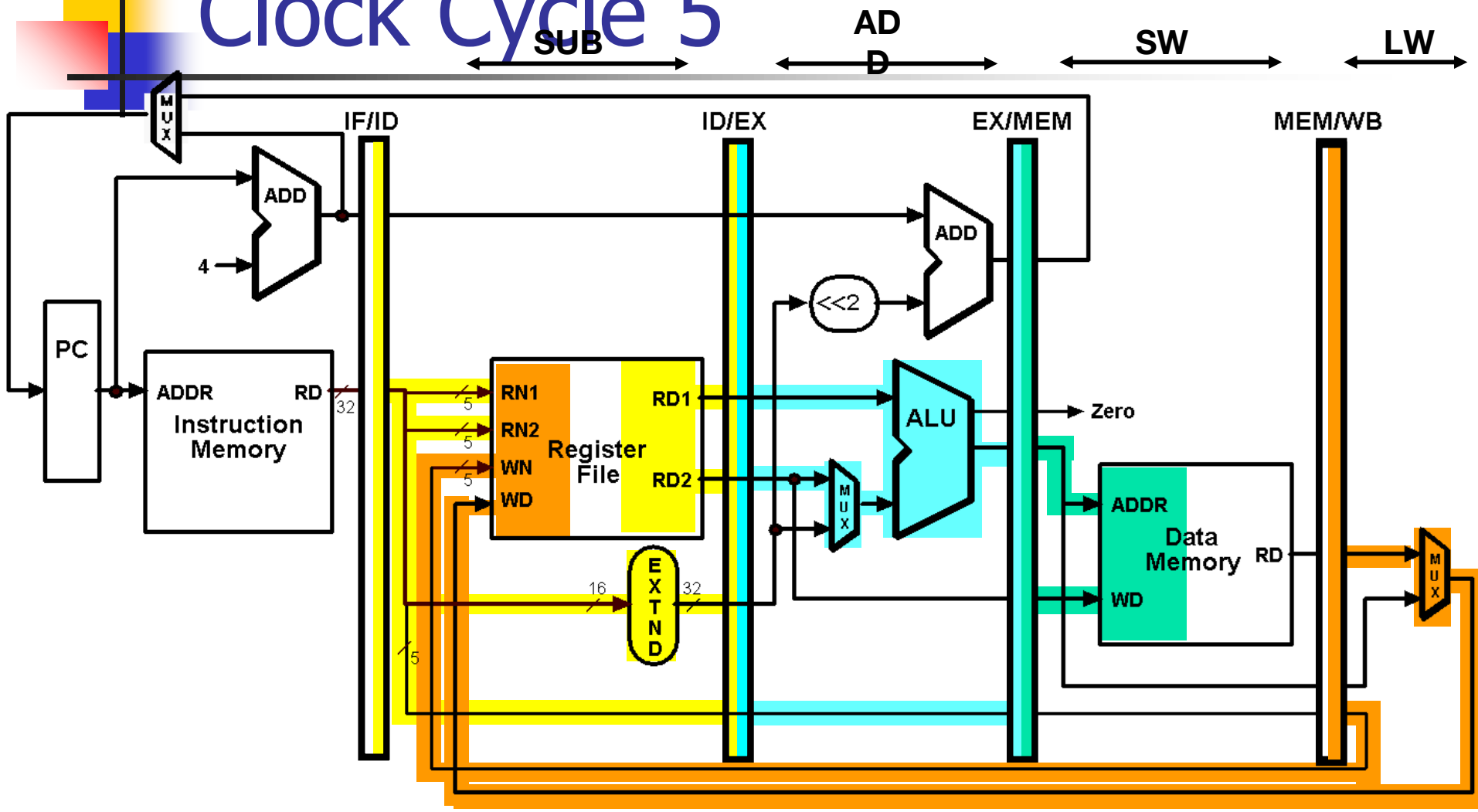
Single-Clock-Cycle Diagram: Clock Cycle 3



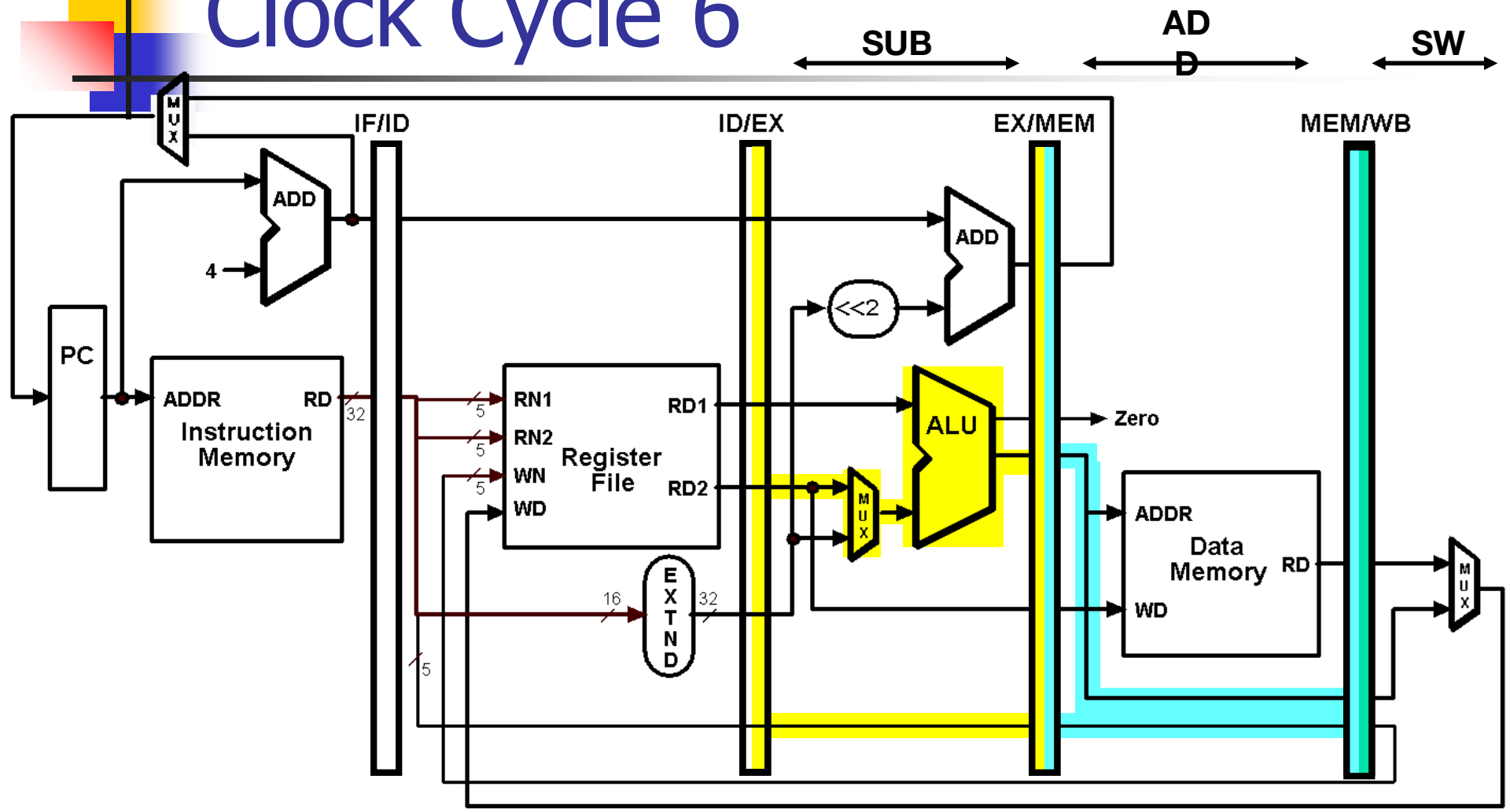
Single-Clock-Cycle Diagram: Clock Cycle 4



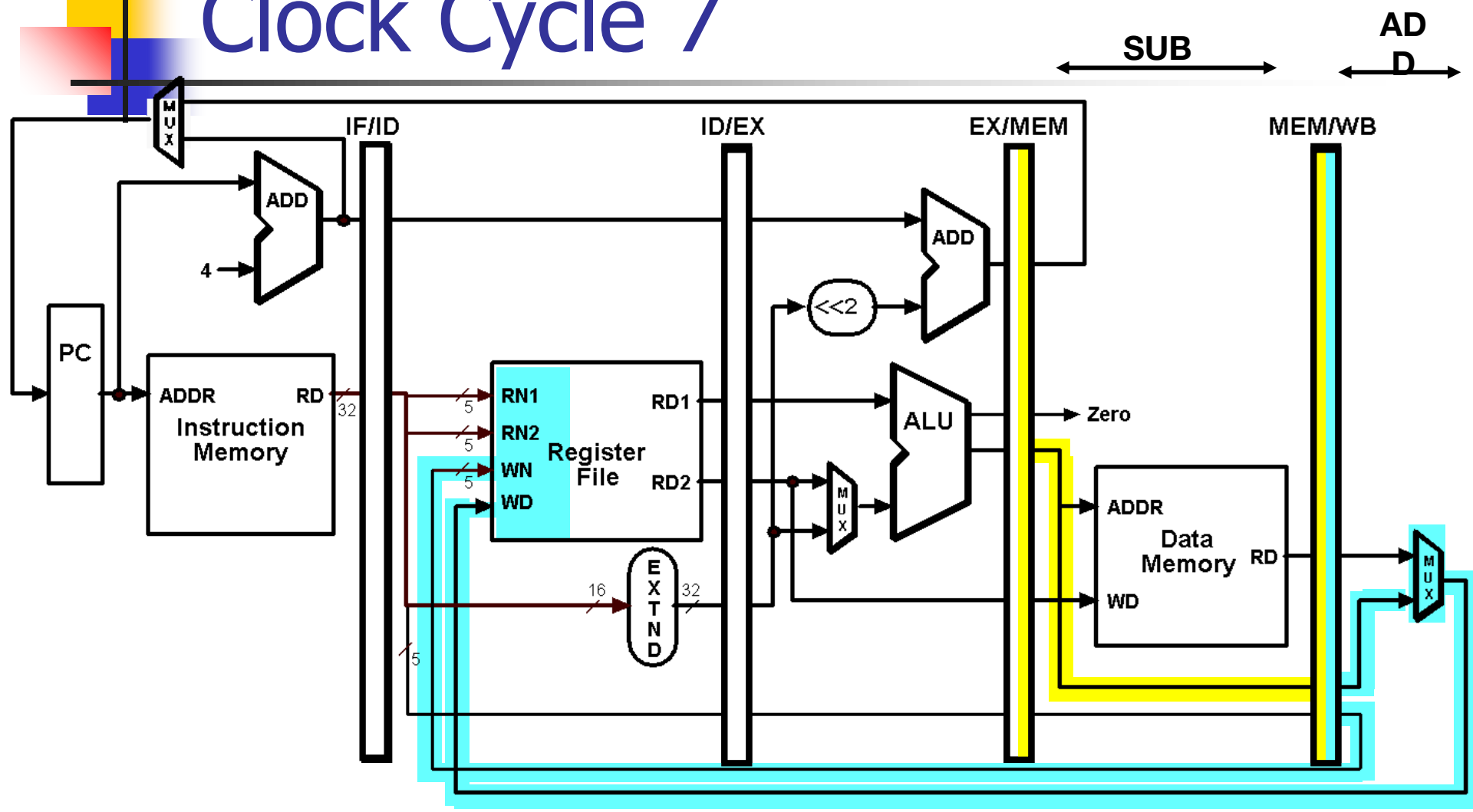
Single-Clock-Cycle Diagram: Clock Cycle 5



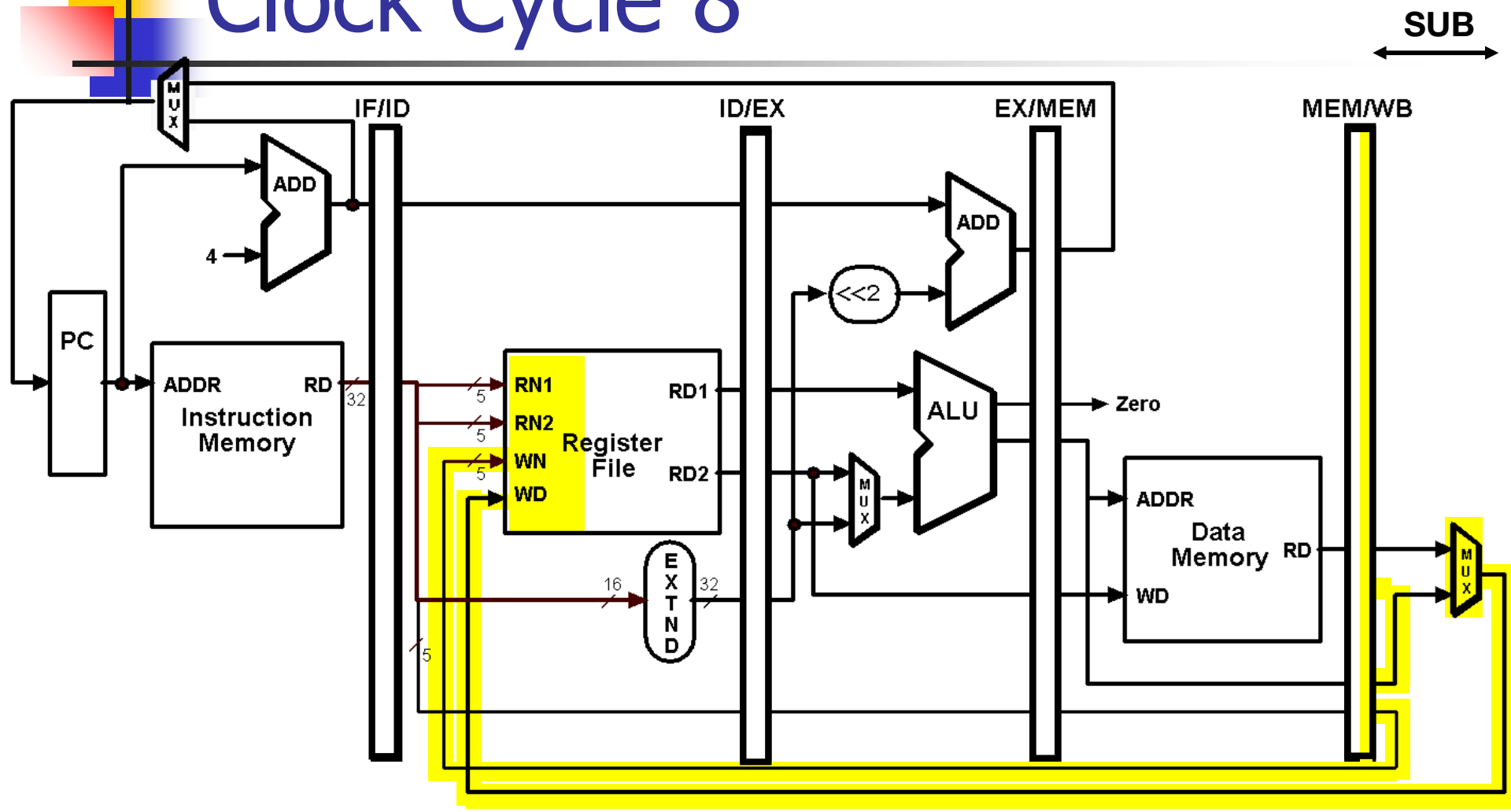
Single-Clock-Cycle Diagram: Clock Cycle 6



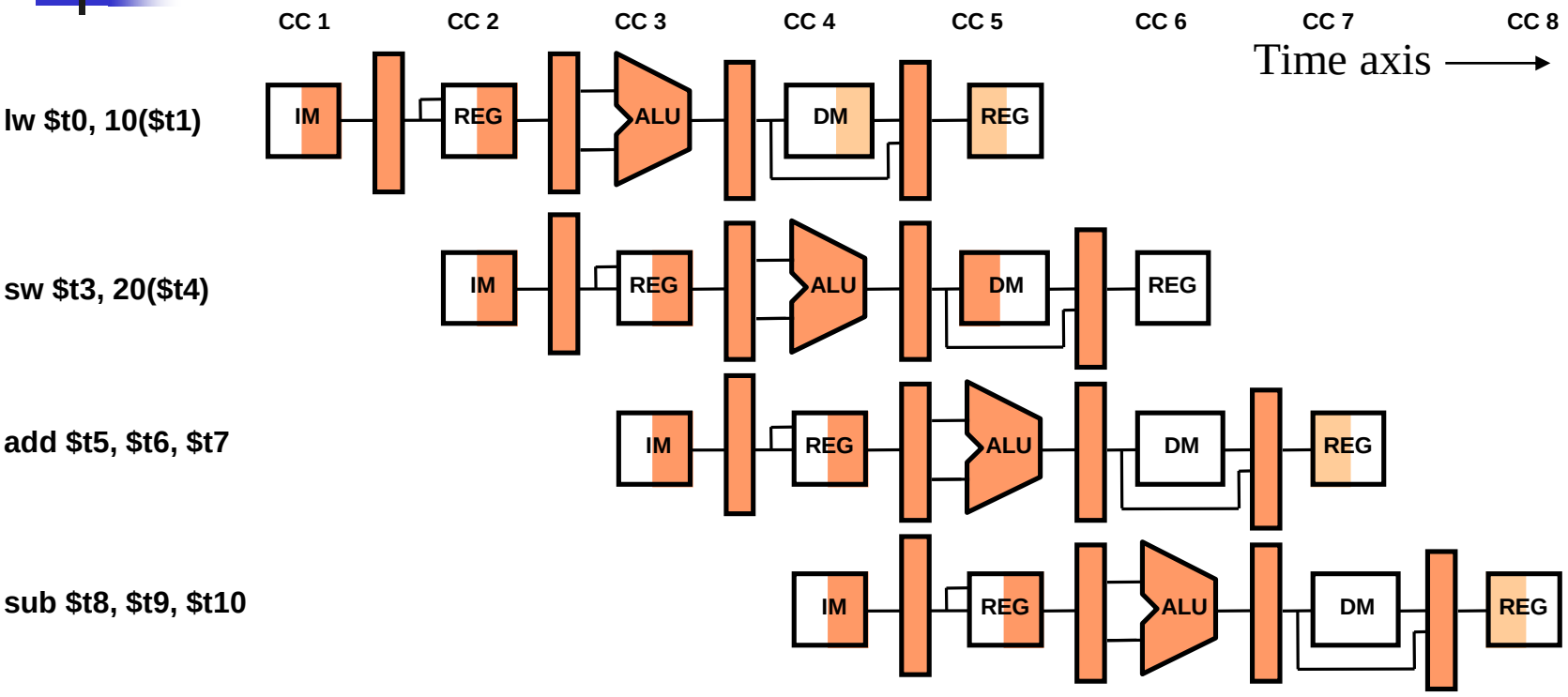
Single-Clock-Cycle Diagram: Clock Cycle 7



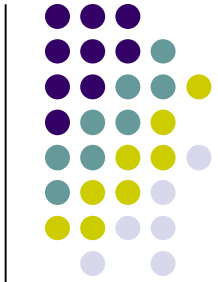
Single-Clock-Cycle Diagram: Clock Cycle 8



Alternative View – Multiple-Clock-Cycle Diagram

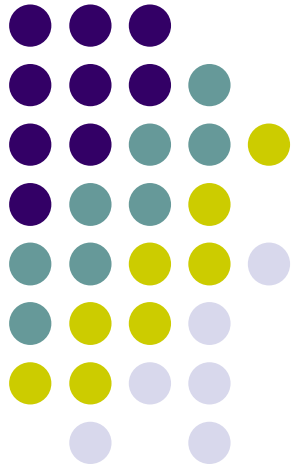


Overview

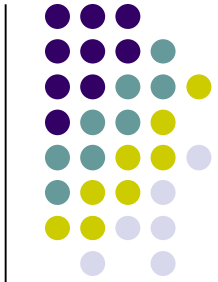


- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts



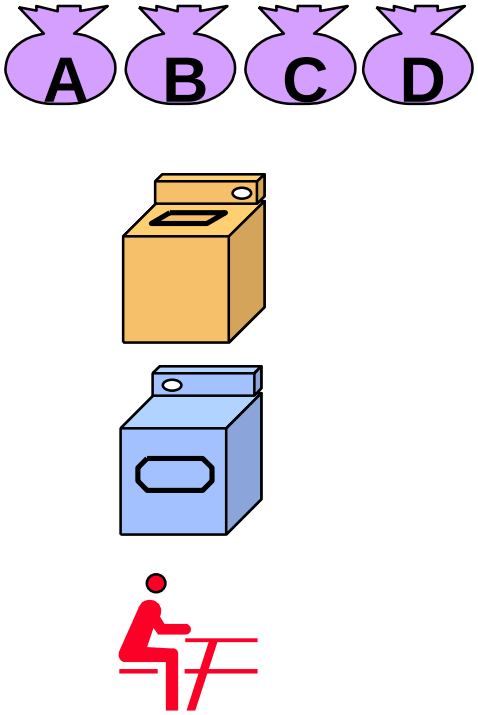
Making the Execution of Programs Faster



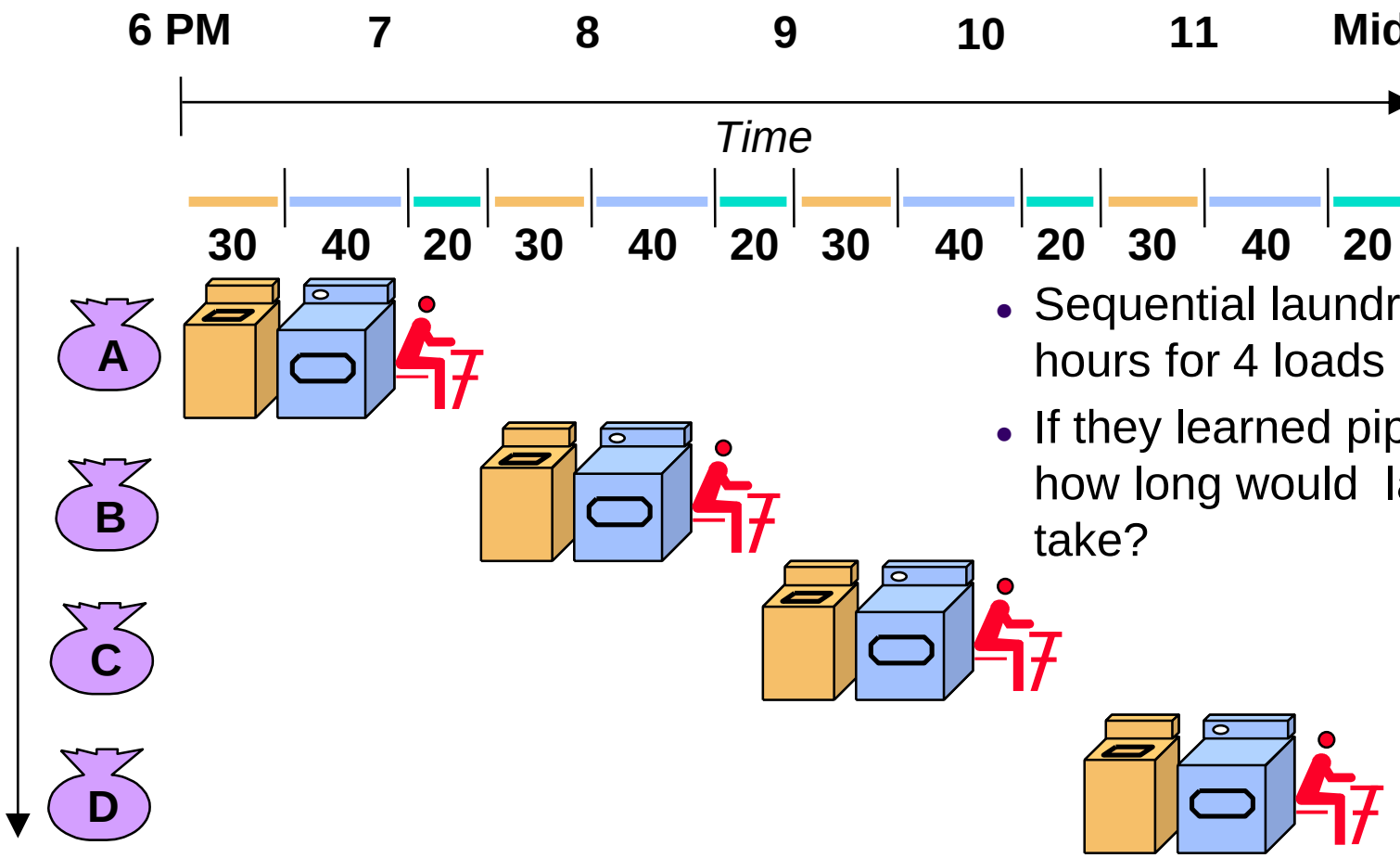
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

Traditional Pipeline Concept

- 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

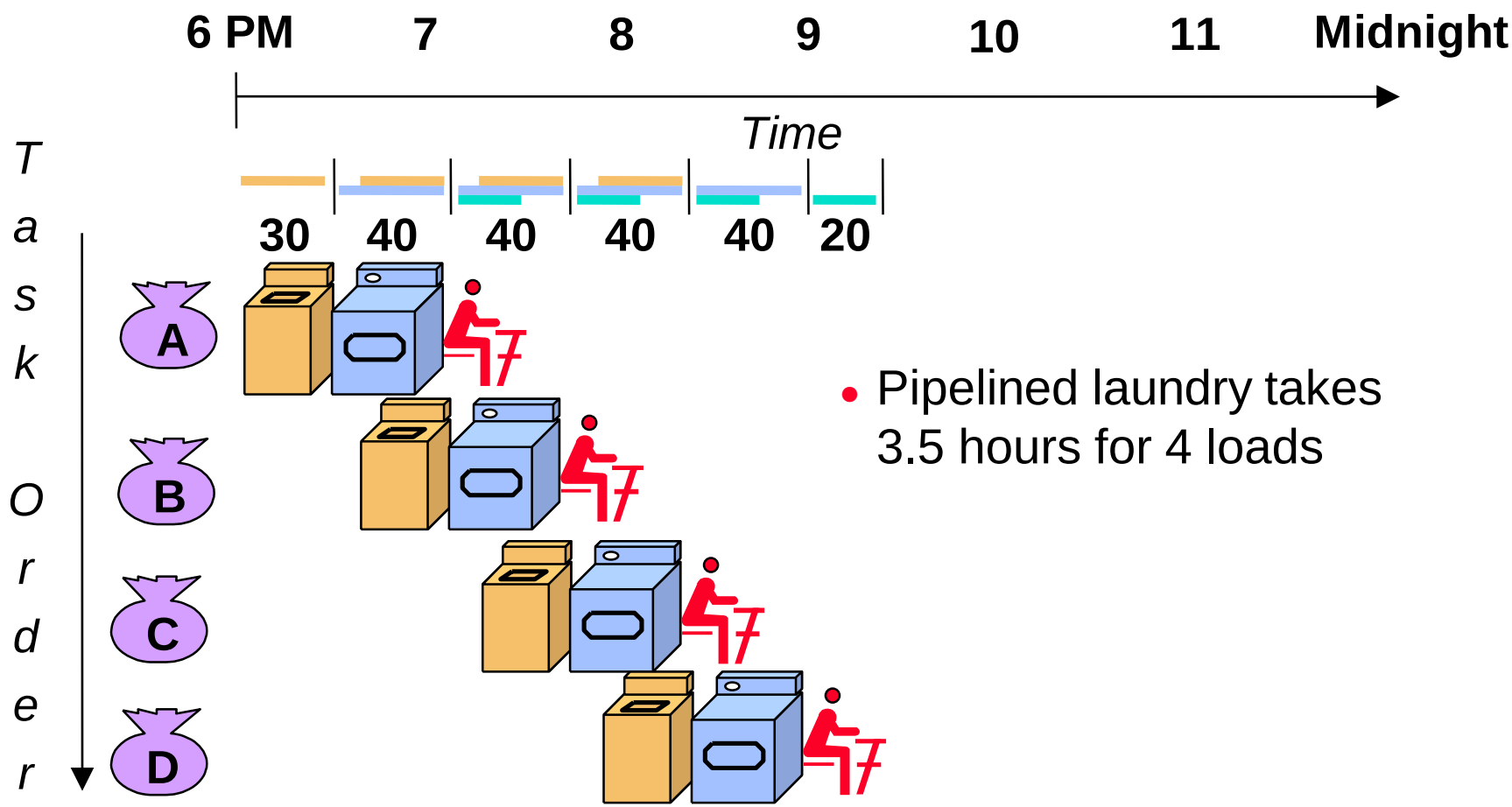


Traditional Pipeline Concept



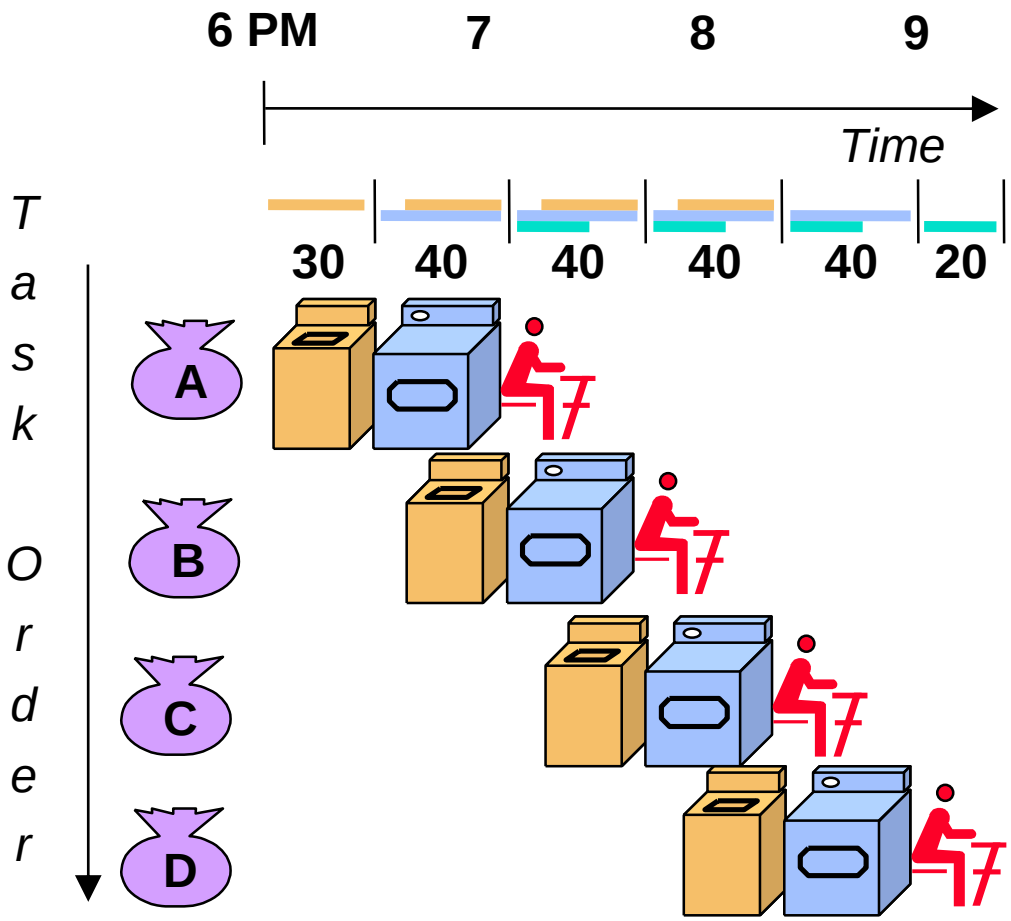
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Traditional Pipeline Concept



- Pipelined laundry takes 3.5 hours for 4 loads

Traditional Pipeline Concept



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Stall for Dependences

Use the Idea of Pipelining in a Computer

Fetch + Execution

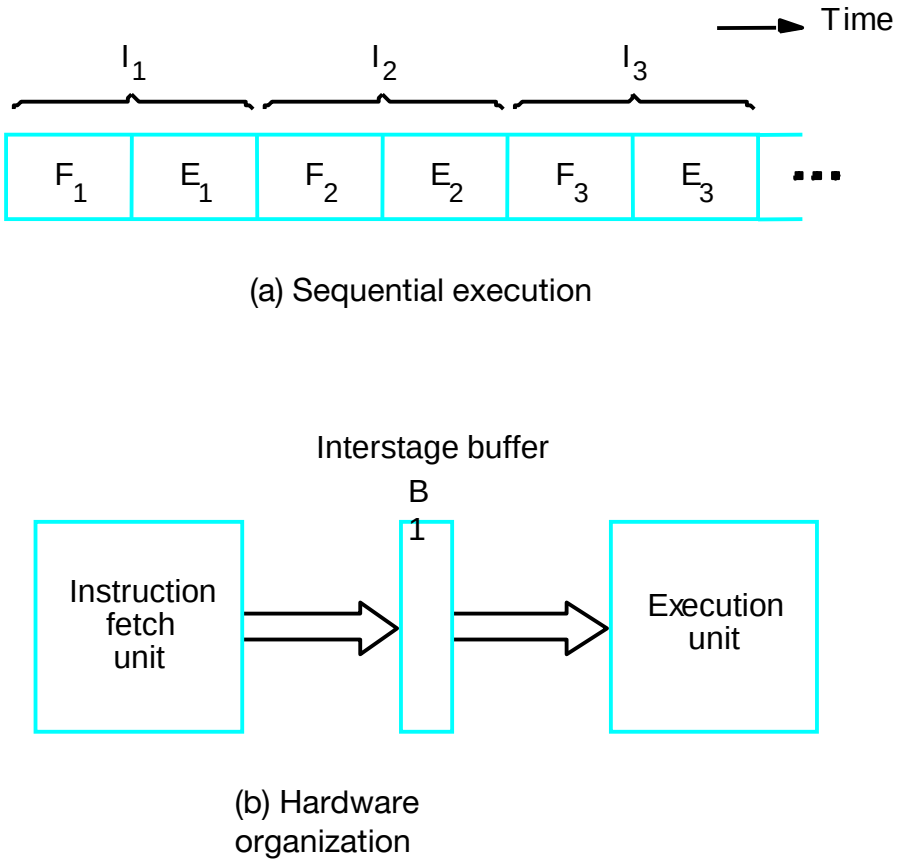
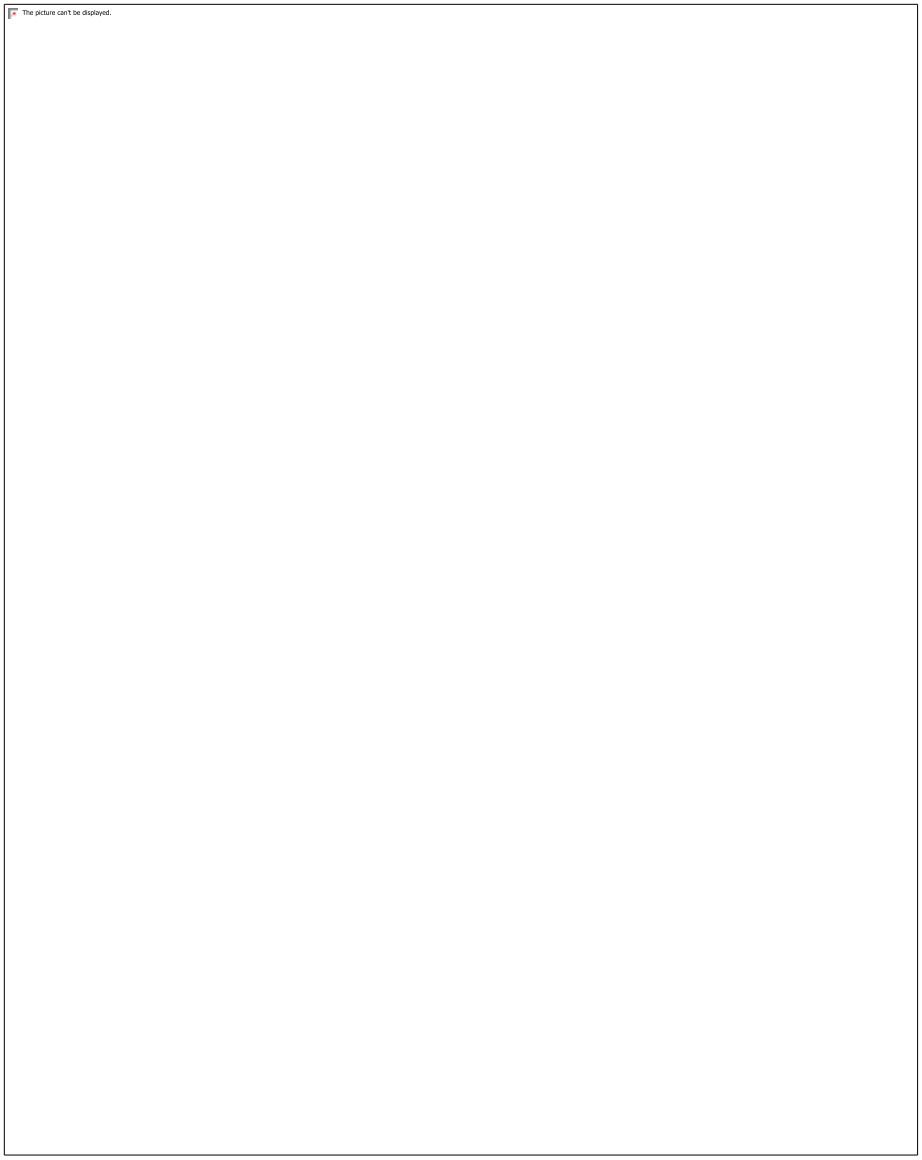


Figure 8.1. Basic idea of instruction pipelining.

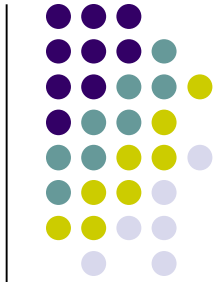
Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write



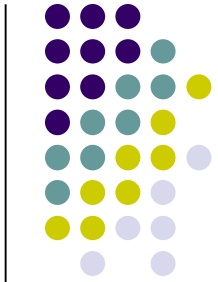
Textbook page: 457

Role of Cache Memory



- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

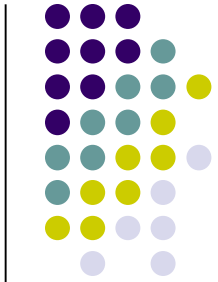
Pipeline Performance



- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

Pipeline Performance

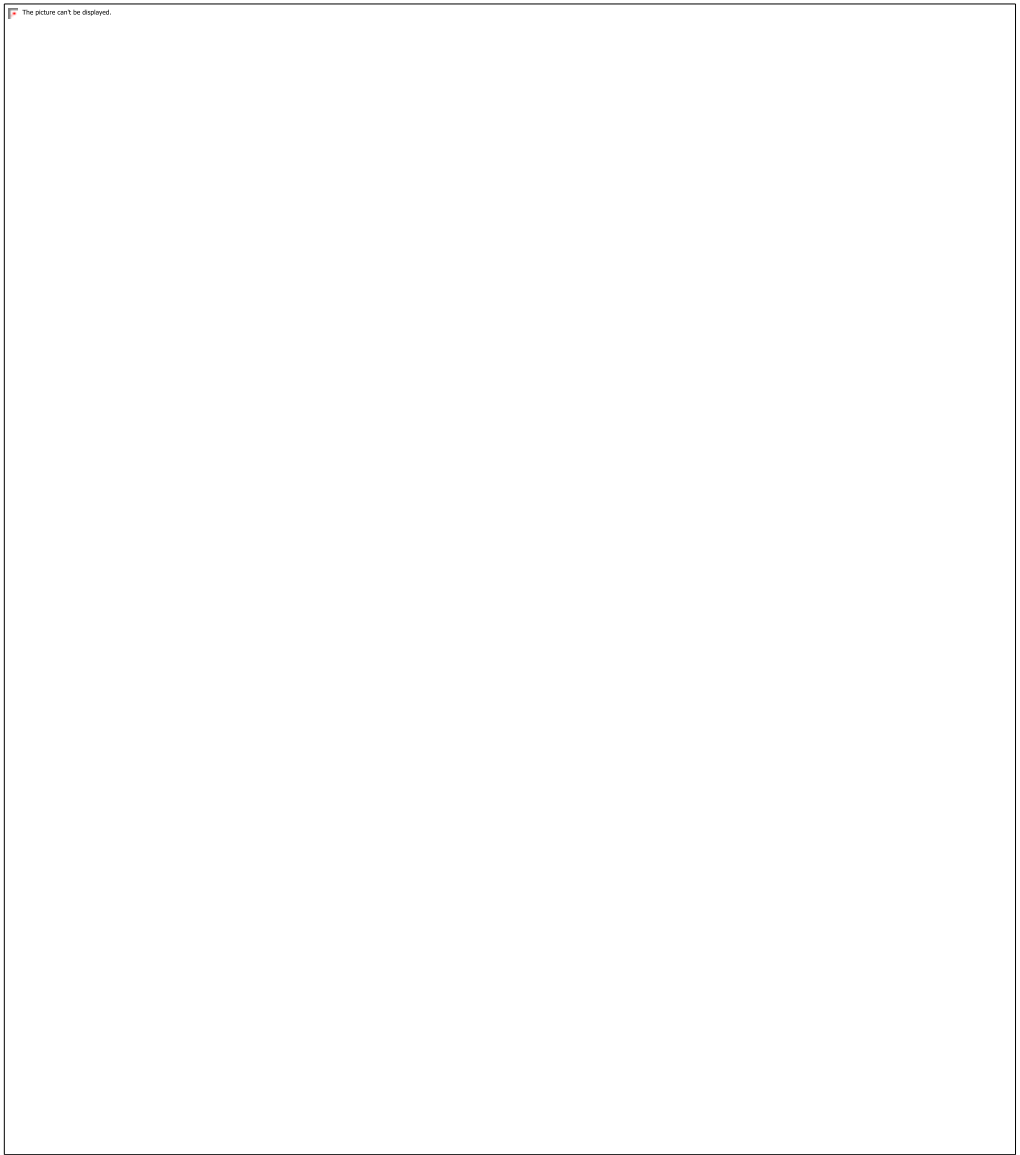
Pipeline Performance



- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

Pipeline Performance

Instruction hazard



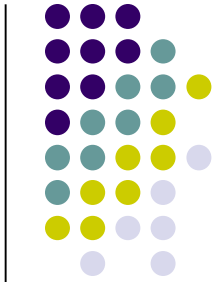
Idle periods – stalls (bubbles)

Pipeline Performance

Structural hazard

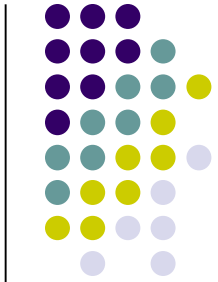
Load X(R1), R2

Pipeline Performance



- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

Quiz



- Four instructions, the I2 takes two clock cycles for execution. Pls draw the figure for 4-stage pipeline, and figure out the total cycles needed for the four instructions to complete.

Pipeline: Introduction

These slides are derived from:
CSCE430/830 Computer Architecture course by Prof. Hong Jiang and Dave Patterson ©UCB

Some figures and tables have been derived from :
Computer System Architecture by M. Morris Mano

Pipelining Outline

Introduction

Defining Pipelining

Pipelining Instructions

Hazards

- Structural hazards
- Data Hazards
- Control Hazards

What is Pipelining?

A way of speeding up execution of instructions

Key idea:

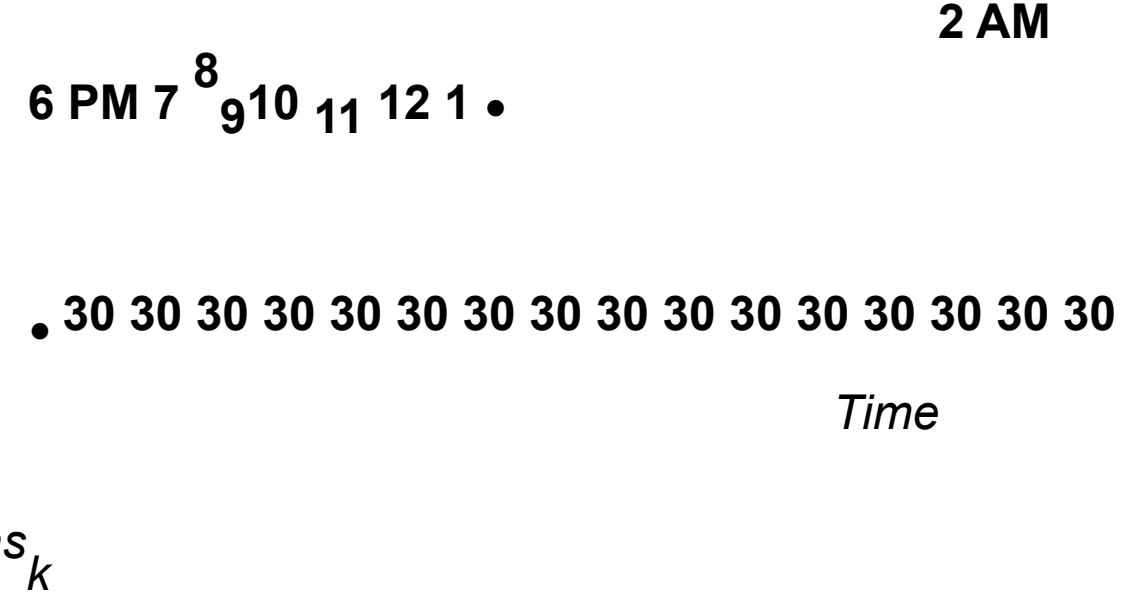
overlap execution of multiple instructions

The Laundry Analogy

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers

If we do laundry sequentially...



Order

To Pipeline, We Overlap Tasks

6 PM 7 8 9 10 11 12 1 2 AM *Time*

30 30 30 30 30 30 30 *Task*

A

B

Order

C

D

- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup

Pipelining a Digital System

-
- Key idea: break big computation up into pieces

  1ns

  Separate each piece with a pipeline register
-

Pipelining a Digital System

Why do this? Because it's faster for repeated computations

Non-pipelined:
1 operation finishes every 1ns

1ns

Pipelined:
1 operation finishes every 200ps

Comments about pipelining

Pipelining increases throughput, but not latency Answer available every 200ps, BUT

-A single computation still takes 1ns

Limitations:

-Computations must be divisible into stage size -Pipeline registers add overhead

- Suppose we need to perform multiply and add operation with a stream of numbers
- $A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \ldots, 7$.

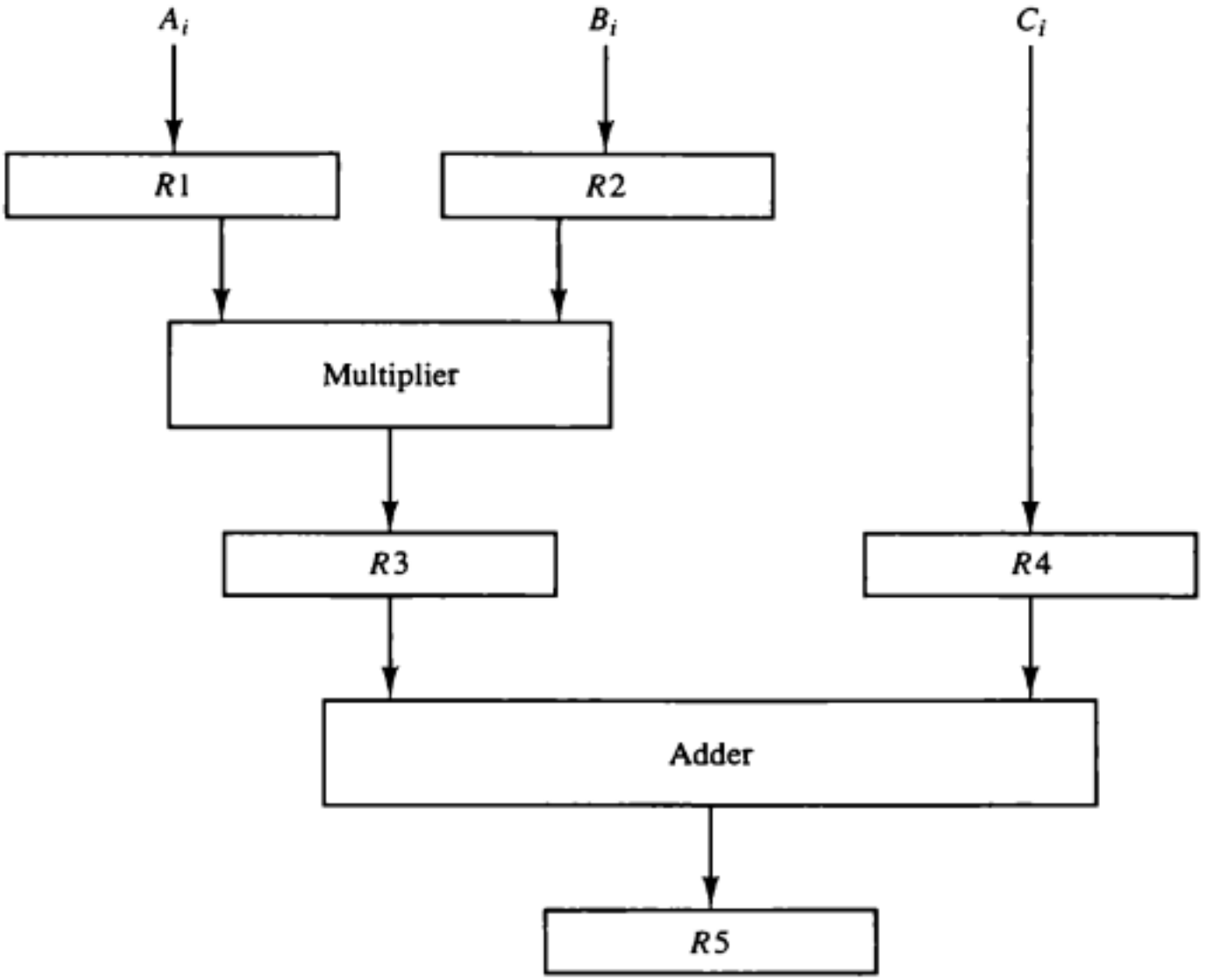
Example of Pipeline Processing

Each subinstruction is implemented in a segment within the pipeline. Each segment has one or two regsiters and a combinational circuit

- The sub operations performed in each segement are as follows

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

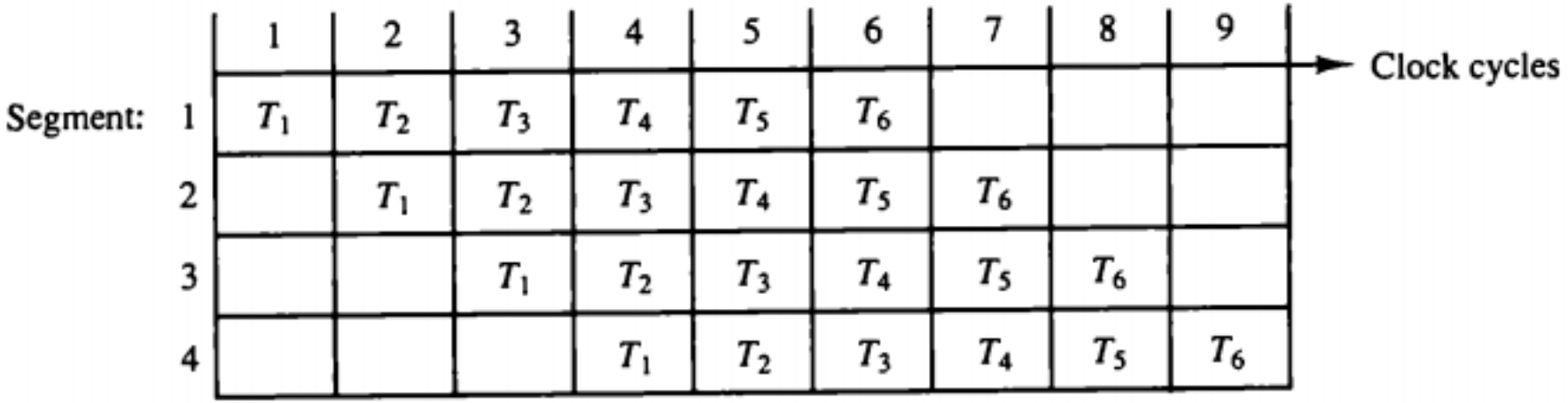
- .
- .


A_i
B_i
C_i
R1
R2
Multiplier
R3
R4
Adder
R5

Content of Registers in Pipeline

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Space Time Diagram of Pipeline



Speedup

Speedup from pipeline

= Average instruction time unpiplined/Average instruction time pipelined

Consider a case for k-segment pipeline with a clock cycle time tp to execute n

tasks. The first task T1 requires a time equal to k*tp to complete its operation since there are k segments in pipeline. The remaining n-1 tasks emerge from the pipe at a rate of one task per clock cycle and they will be completed in k+n-1 clock cycles.

Next, to concsider an unpipeline unit that performs the same operation and takes a time equal to tn to complete the task. The total time required fro n tasks is n*tn. The speed up of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

Speedup

- As the number of tasks increase n becomes much larger than k-1, and k+n-1 approaches

$$S = \frac{t_n}{t_p}$$

the value of n. Under this condition, the speed up

becomes .

If we assume the the time taken to process the task is the same as in the pipeline and nonpipeline circuits, we will have $t_n = kt_p$

- The speedup then reduces to numer of stages of pipeline

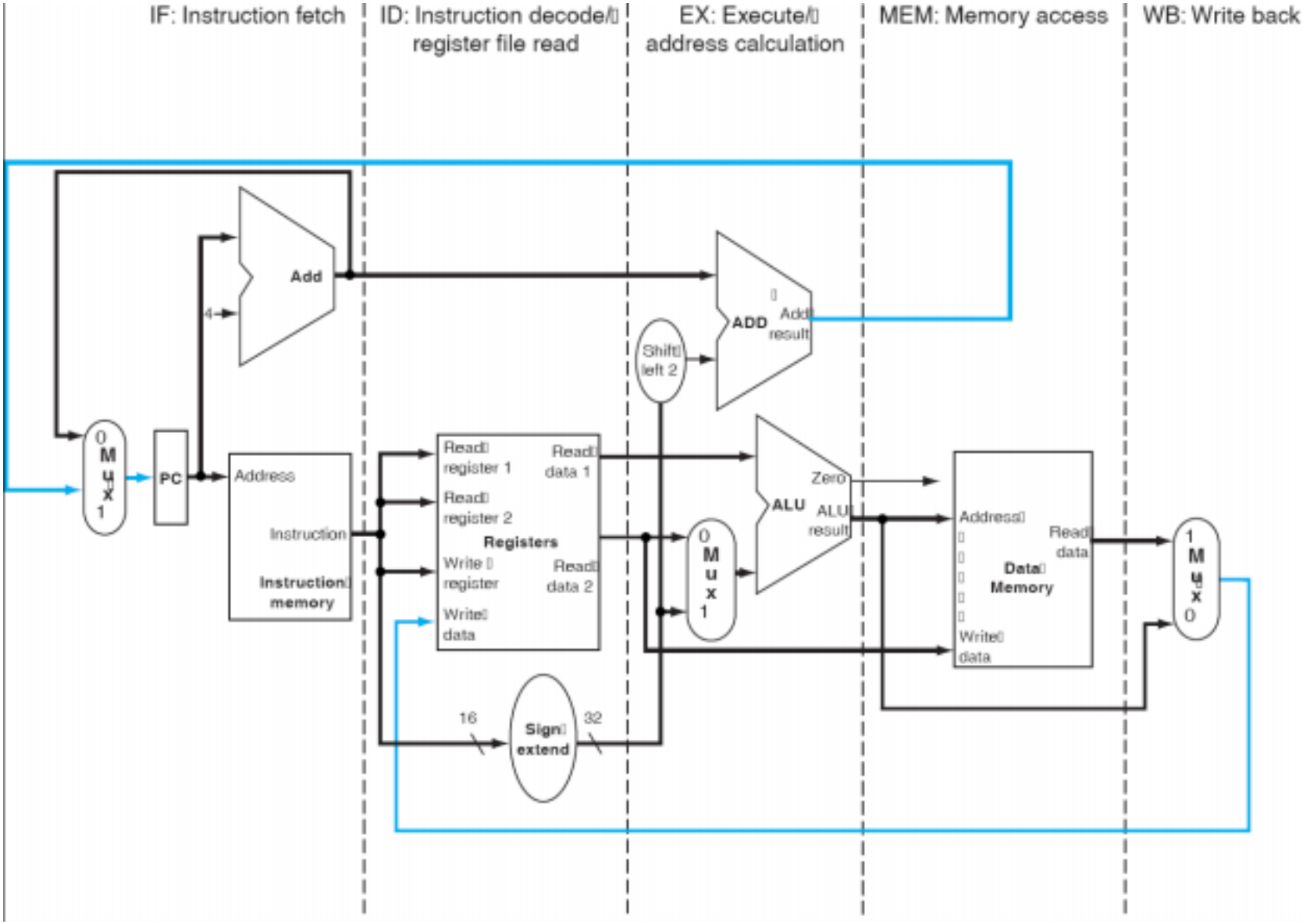
$$S = \frac{kt_p}{t_p} = k$$

Pipelining a Processor

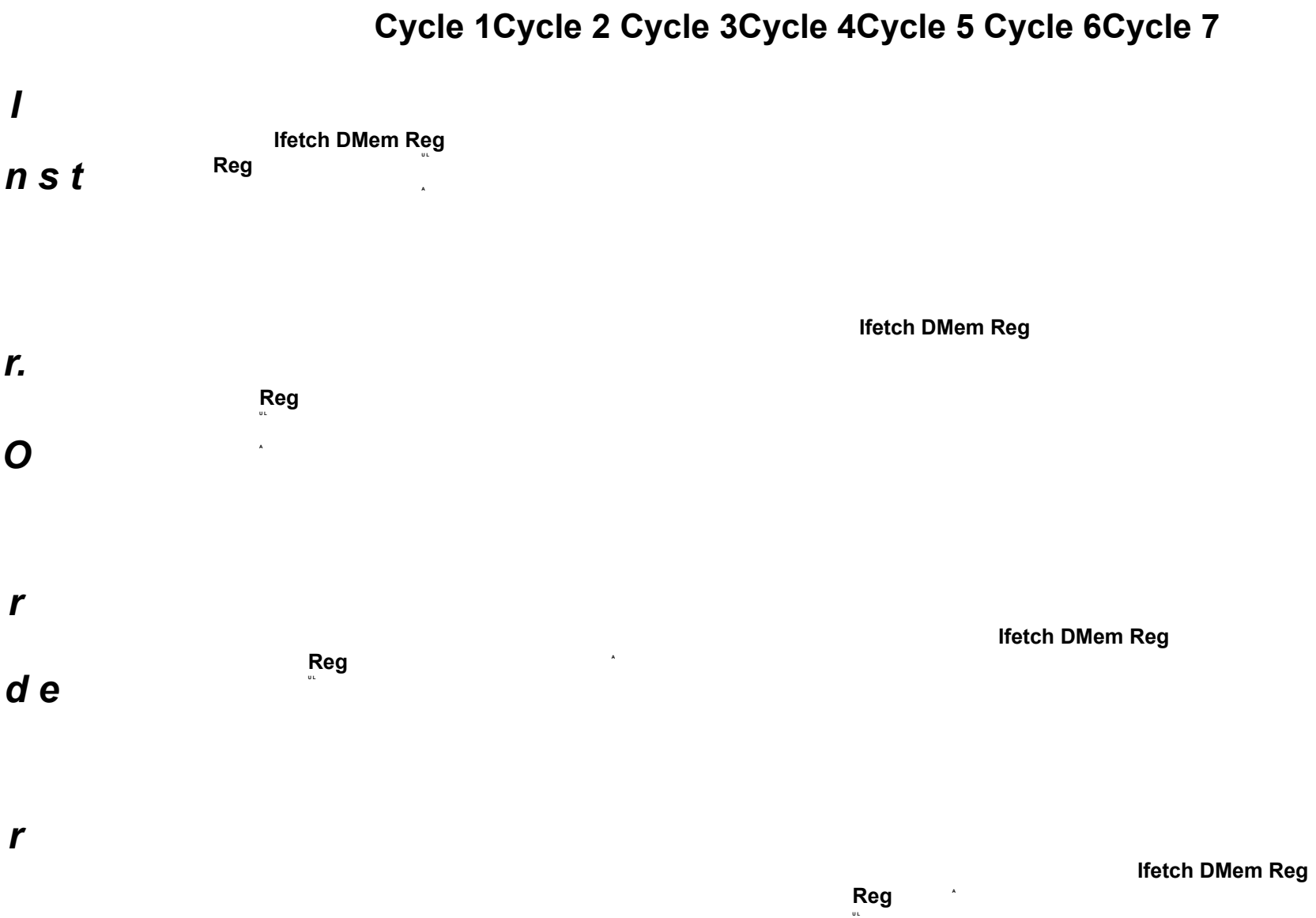
- Recall the 5 steps in instruction execution:
  1. Instruction Fetch (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution operation or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)

- Review: Single-Cycle Processor
  - All 5 steps done in a single clock cycle – Dedicated hardware required for each step

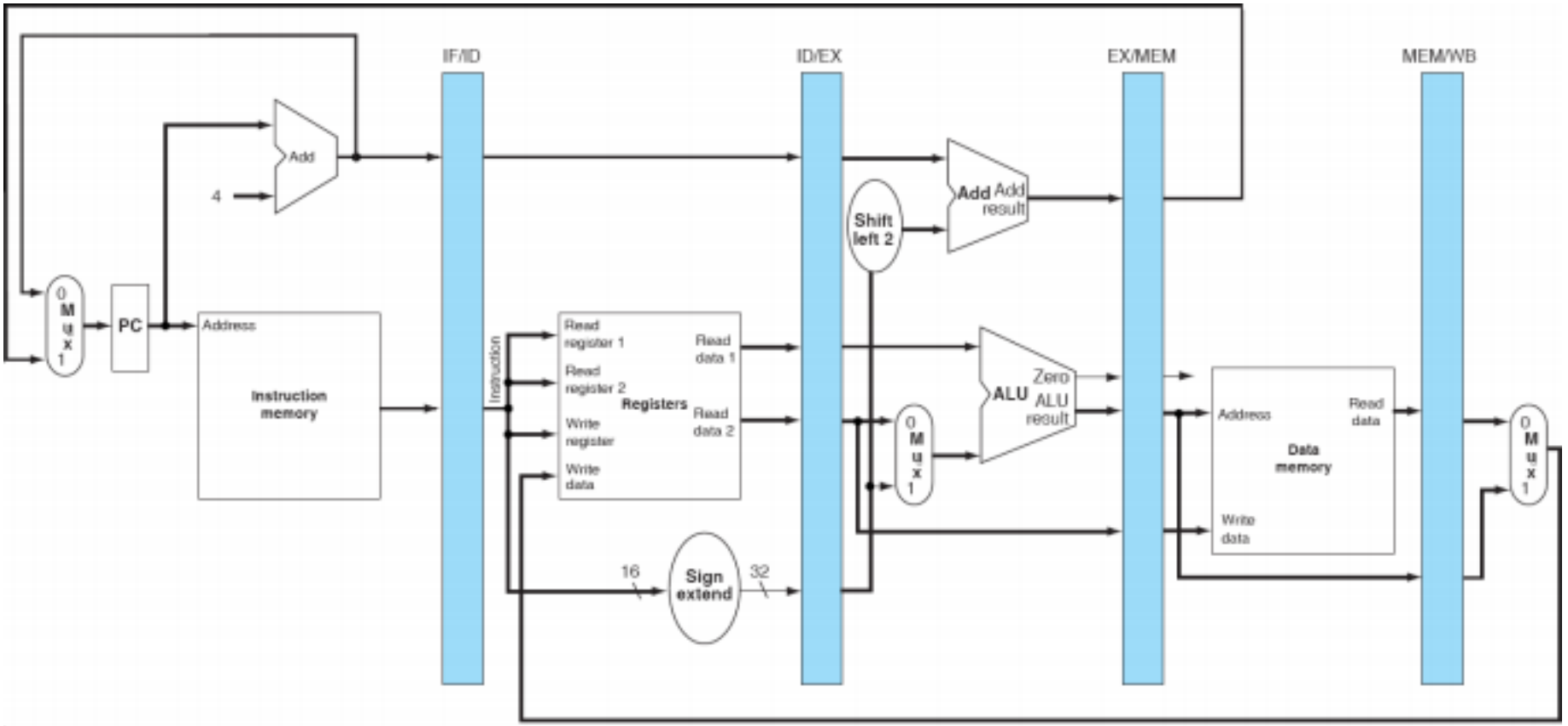
Review - Single-Cycle Processor



The Basic Pipeline For MIPS



Basic Pipelined Processor



Single-Cycle vs. Pipelined Execution

Non-Pipelined

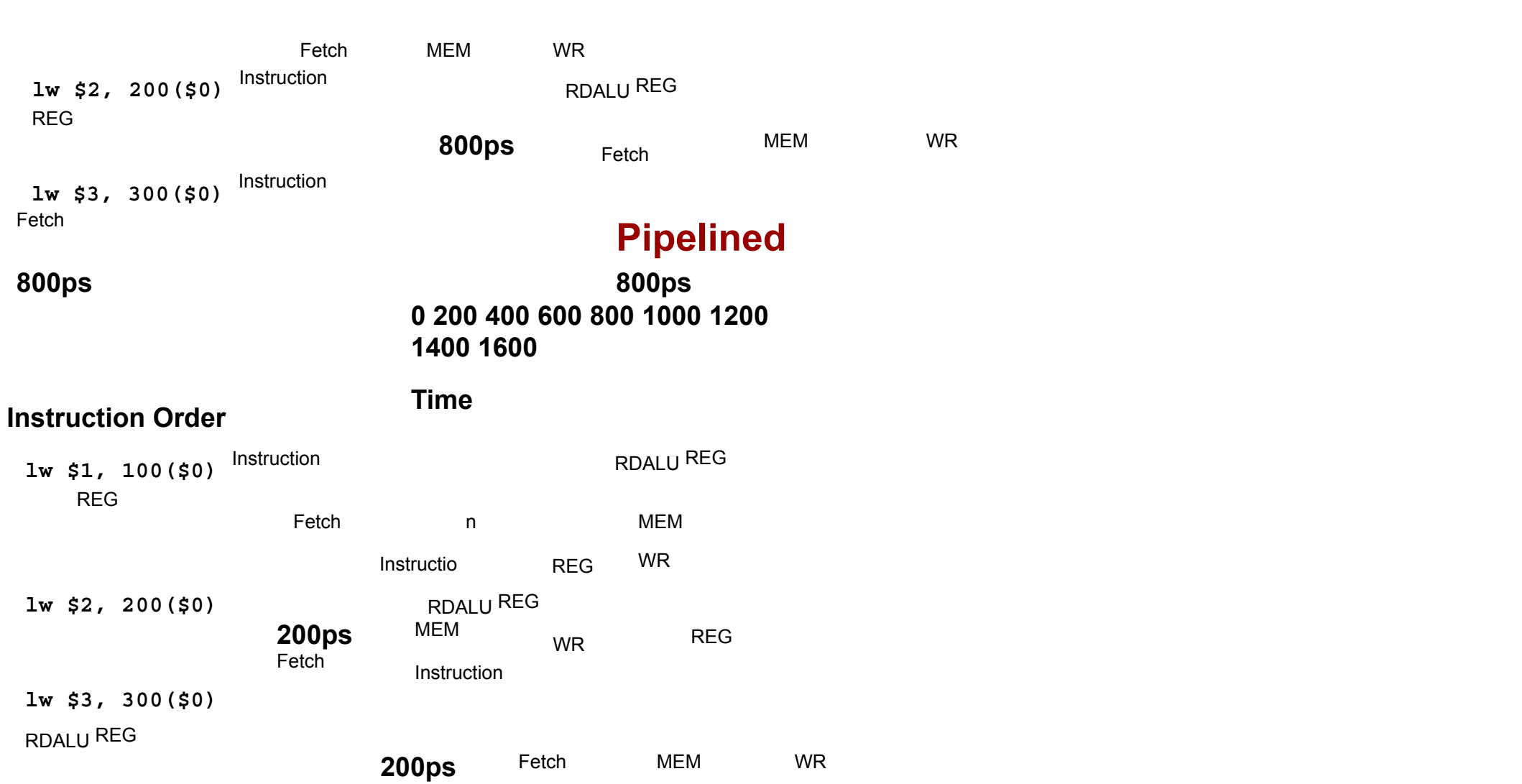
Instruction Order **1600 1800**

0 200 400 600 800 1000 1200 1400

Time

`lw $1, 100($0)` Instruction RDALU REG

REG



Comments about Pipelining

The good news

- Multiple instructions are being processed at same time

- This works because stages are isolated by registers -

Best case speedup of N

The bad news

- Instructions interfere with each other - hazards

  Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle

  Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipeline Hazards

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

Structural hazards: two different instructions use same h/w in same cycle

Data hazards: Instruction depends on result of prior instruction still in the pipeline

Control hazards: Pipelining of branches & other instructions that change the PC

Objective(s)

- One of the objectives of this study is to make each student to familiar with the pipeline architecture of a commercial Microprocessor such as MIPS system.
- Once the students gain the knowledge of the pipelined hardware structure of a microprocessor, then it is possible for them to apply that into any microprocessor related fields.
- Finally, the students will get the knowledge of the complete operation a computer system.

Introduction

- This section describes the features of a **five-stage RISC**(Reduced Instruction Set Computer) **pipeline machine – MIPS 32-bit system** and its issue of hazards and performance problems.

What is Pipelining?

- **Pipeline** is a performance improvement technique- **multiple instructions are overlapped in execution**
  - Pipeline takes advantages of **parallelism** that exists among the actions needed to execute an instruction
    - **Today, pipelining is the key performance technique used to make fast CPUs**
- A pipeline is like an automobile assembly line
- In a ***computer pipeline***, each step in the pipeline completes a part of an instruction, each of these steps called a ***pipe stage*** or a ***pipe segment***
  - The stages are connected one to the next to form a pipe

Pipelining

- In an automobile assembly line, **throughput** is defined as the ***no. of vehicles per hour***
- The ***throughput*** of an instruction pipeline is determined by ***how often an instruction exits in the pipeline***
  - Because the pipeline stages are hooked together, all the stages must be ready to proceed at the same time

- The time required between moving an instruction from one stage to the next of the pipeline is a ***processor cycle*** (or a ***pipeline cycle***)
  - Length of a *processor cycle* is determined by the time required for the slowest pipe stage

Pipelining

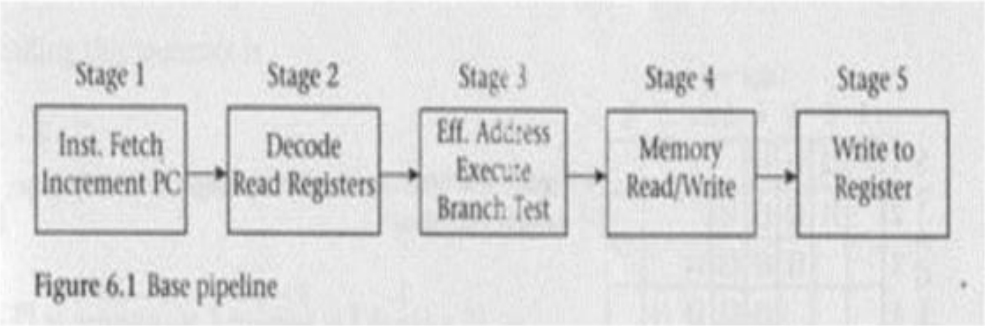


Figure 6.1 Base pipeline

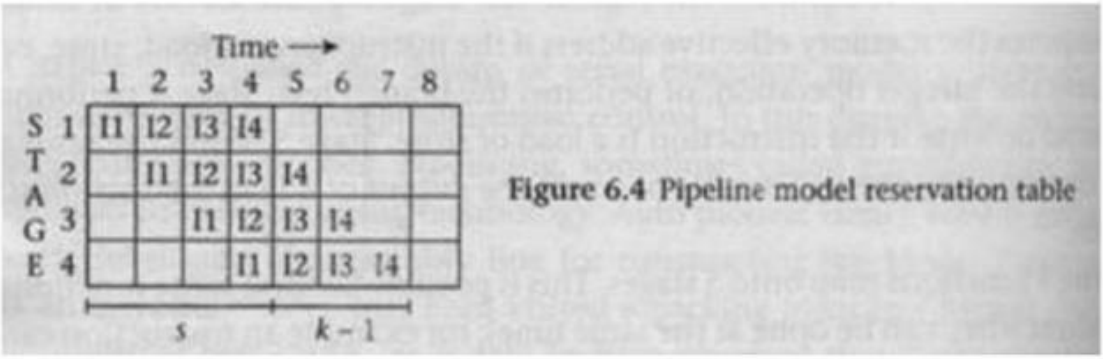


Figure 6.4 Pipeline model reservation table

Goal of Pipelining Designer

- Goal of a pipeline designer is to balance the length of each pipeline stages
- If the stages are perfectly balanced, then the ***time per instruction*** on the pipelined processor is

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipeline stages in a pipelined machine}}$$

- The ***speedup*** from pipelining equals to the **no. of pipe stages** (only when the pipeline CPI is 1)

Goal of Pipelining Designer

- Pipelining yields a ***reduction in the average execution time per instruction***
  - *Pipelining does not reduce the execution time of an instruction*
  - The reduction can be viewed as decreasing
    - **CPI,**
    - **Clock cycle time,**
    - **Or combination of both**
- Pipelining is an implementation technique that **exploits parallelism among the instructions** in a sequential instruction stream
  - **It is not visible to the programmer**

The Basic MIPS Instruction Set

- **MIPS** architectures are characterized by the following few key properties:
  - **All operations** on data apply to data in registers and typically change the entire register (32 bits/reg in a MIPS-32bit system)
  - The only operations that affect memory are ***load*** and ***store*** operations
    - Load and store operations that load or store a full register
  - **Size** of an instruction is fixed
- These properties simplifies the implementation of pipelining in **RISC** processors (MIPS is a RISC processor)

Five Stages of MIPS Pipeline Unit

- **IF** (Instruction fetch) **stage**:
  - Send the PC (Program Counter) to memory and fetch the current instruction from memory
  - Update the PC for the next sequential instruction by adding 4 to the PC (each instruction is 4byte in size)
- **ID** (Instruction decode/register fetch):
  - *Decode* the instruction and *read the operand registers* from the register file
  - Do the *equality test* on the registers, for branch test
  - *Compute the branch target address* by adding the offset to the incremented PC

A simple Implementation of a RISC Instruction Set

- **Instruction decoding done in parallel with reading registers**, because register specifiers are at a fixed location in a RISC architecture, called ***fixed-field decoding***

- **EX** (Execution/effective address cycle):
  - **Memory reference** operation: ALU adds the *base register* and the *offset* to form the effective address for load/store
  - **Register-Register** operation: ALU performs the operation specified by the ALU opcode on the value given by register operands
  - **Register-Immediate** operation: ALU performs the operation specified by the ALU opcode based on the immediate value

A simple Implementation of a RISC Instruction Set

- **MEM** (Memory access) stage:
  - If the operation is a ***load***, memory read uses the **effective address** which is computed in the previous cycle
  - If it is a ***store***, the memory writes happen using the effective address
- **WB** (Write-back result to register file) stage:
  - WB stage is supports only **reg-reg** and **load** instructions
  - Write the result into the register file, whether is from memory (load) or from ALU

Classic Five-stage Pipeline for a RISC Processor

- Each of the clock cycle from previous section becomes a *pipe stage* – a cycle in the pipeline
  - Pipe stages shown in **Figure A.1**
- Although each instruction takes **5 clock cycles** to complete (each stage needs 1 cock cycle), during each clock cycle the HW will initiate new instruction

Classic Five-stage Pipeline for a RISC Processor

- Each of the clock cycle from previous section becomes a *pipe stage* – a cycle in the pipeline
  - Pipe stages shown in **Figure A.1**
- Although each instruction takes **5 clock cycles** to complete (each stage needs 1 cock cycle), during each clock cycle the HW will initiate new instruction

Classic Five-stage Pipeline for RISC Processor

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

Figure A.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

Classic Five-stage Pipeline for RISC Processor

- In a pipeline, we don't try to perform two different operations with the same **data path** resource on the same clock cycle
  - *A single ALU **cannot be asked** to compute an effective address and perform a subtract operation at the same time*
  - *we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict*
  - **Figure A.2** shows a simplified version of a **RISC data path drawn in pipeline fashion**

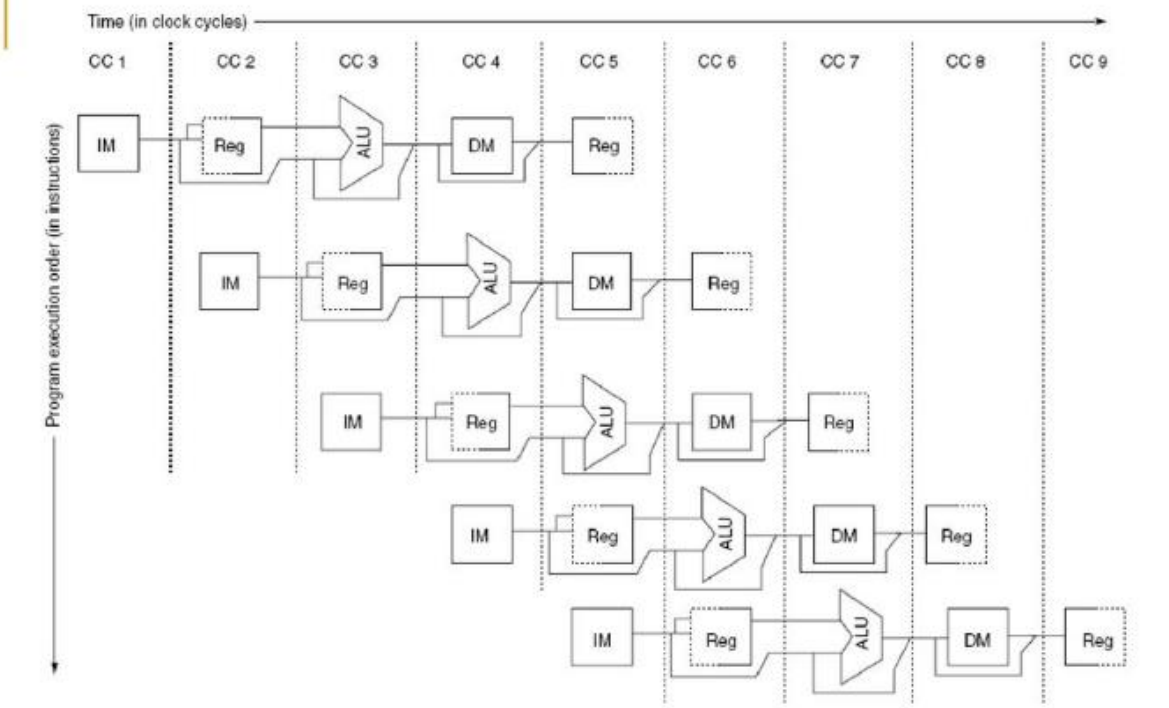


Figure A.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice.

Classic Five-stage Pipeline for RISC Processor

- **Figure A.2** showed that the major functional units are used in different cycles, and hence **overlapping the execution of multiple instructions causes a few conflicts**
- There are **three observations**:
  - **First-** The use of **separate instruction and data caches** eliminates a conflict (*resource conflict*) for a single memory for instruction and data accesses
  - **Second-** The register file is used in the two stages: *one for reading in ID stage and one for in WB stage*
    - **Need to perform two reads and one write every clock cycle**
  - To handle reads and a write to the same register, perform **write in the first half and the read in the second half of the clock cycle**

Classic Five-stage Pipeline for RISC Processor

- **Third** – To start a new instruction every clock cycle, increment and store PC every clock in the IF stage
  - **Must keep an adder in the ID stage to compute the potential branch target**
- Must insure that instructions in different stages of the pipeline do not interfere with one another- **HW resource conflict**

- Hence each stage of the pipeline is separated by introducing **pipeline registers** (**Figure A.3**)
  - So that at the end of a clock cycle all the results from a given stage are stored into **a register** (buffer) that is used as the input to the next stage on the next cycle

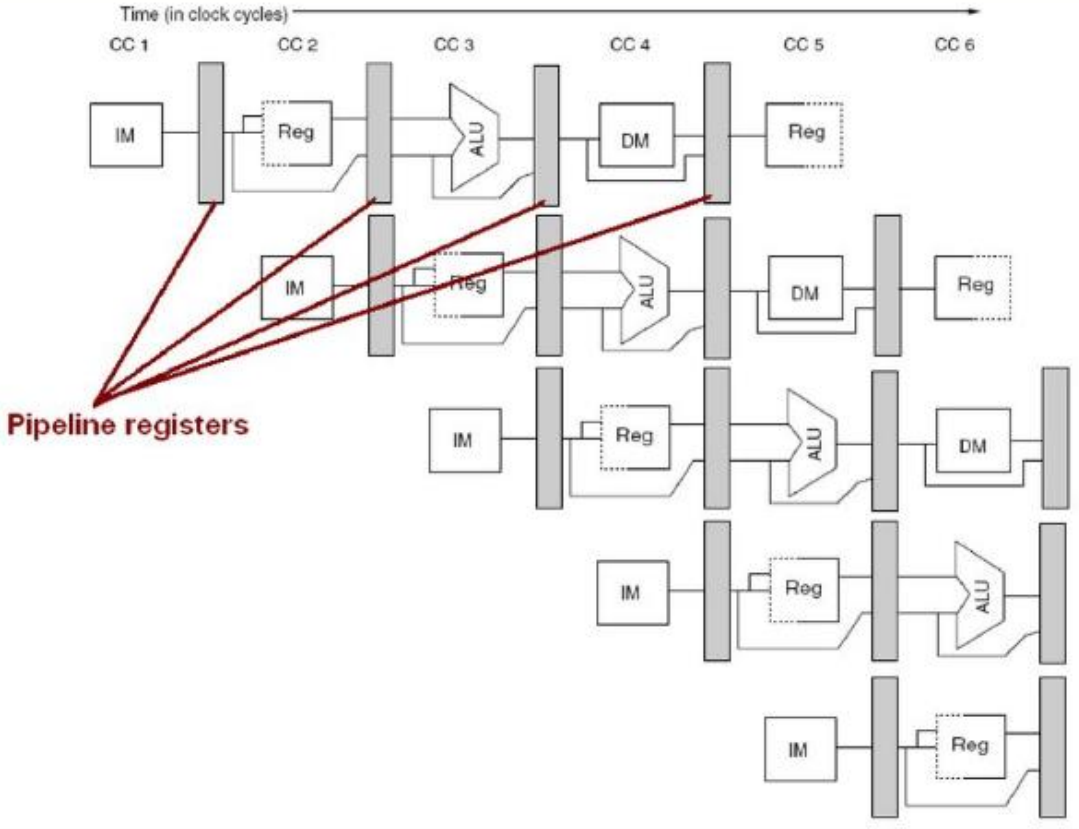


Figure A.3 A pipeline showing the pipeline registers between successive pipeline stages.

Classic Five-stage Pipeline for RISC Processor

- The **pipeline registers** play the key role of carrying ***intermediate results*** from one stage to another where the source and destination may not be directly adjacent
  - For ex. the register value to be stored during a store instruction is read during ID, but not actually used until MEM;
  - It is passed through two pipeline registers to reach the data memory during the MEM stage
  - Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers
- Look at the details of a four-stage pipeline to get a clear pipeline idea!

Basic Performance Issue in Pipelining

- Pipelining **increases CPU's instruction throughput**
  - ***The no. of instructions completed per unit time*** increases
- Pipeline **does not reduce the execution time** of an individual instruction
- But it **increases the exe. time of each instruction** due to overhead in the control of the pipeline mechanism
- **Due to pipeline process**, ***a program runs faster even though no single instruction runs faster***
- Issues arise from pipeline are ***latency***, ***imbalance among pipeline stages*** and ***pipeline overhead***

Basic Performance Issue in Pipelining

- **Imbalance among the pipe stages** reduces performance since the clock cannot run faster than the time needed for the slowest stage (such as memory access)
- **Pipeline overhead** arises from the combination of ***pipeline register delay*** and ***clock skew***
  - That is, ***pipeline registers add setup time and propagation delay*** **to the system**
- **Clock skew** is the maximum delay between when the clock arrives at any two pipeline registers
- In a Pipeline, **average instr. execution time** can be given as:

Average instruction exe. time = Clock cycle time x Average CPI