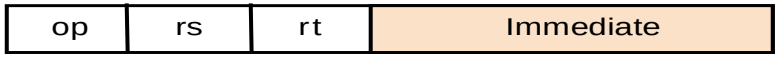
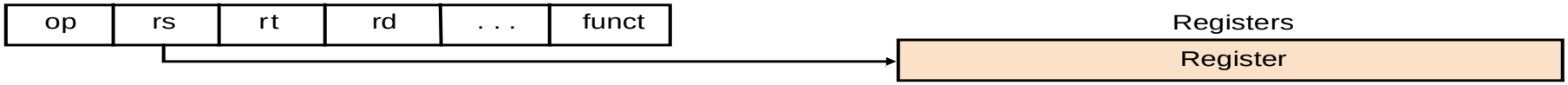
Summary of MIPS Addressing Modes

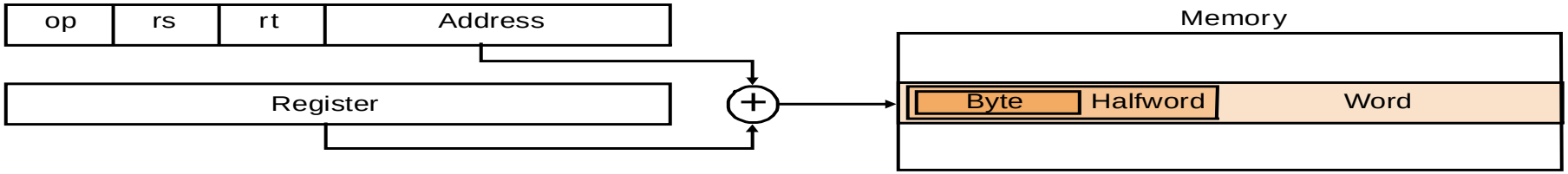
1. Immediate addressing



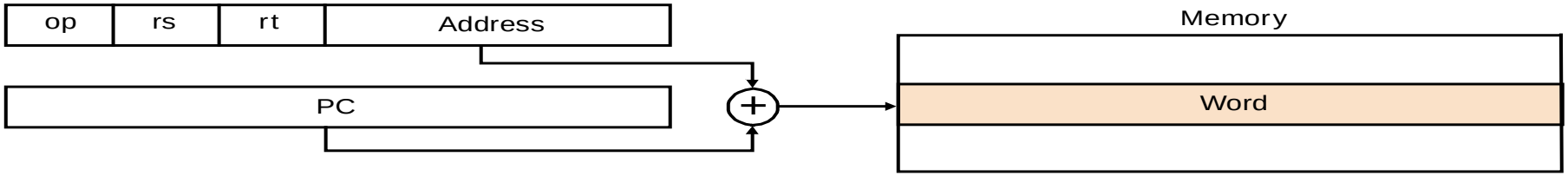
2. Register addressing



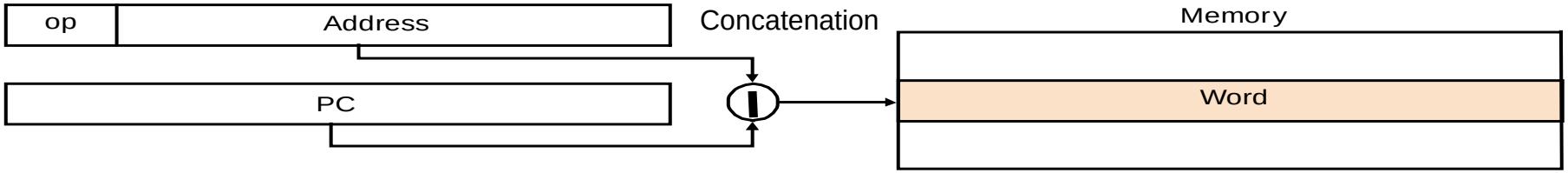
3. Base addressing



4. PC-relative addressing



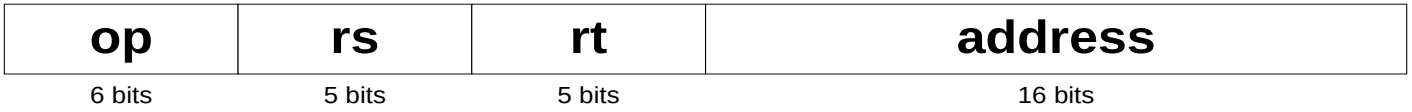
5. Pseudodirect addressing



1010 1101 0001 0000 0000 0000 0000 0100

Opcode

Determine the format



The first 6 bits represent the opcode: 1010 11 = 32 + 8 + 2 + 1= 43.

➔ sw (store word) instruction.

opcode - 6 bits	rs - 5 bits - dest base	rt - 5 bits - source	offset - 16 bits
1010 11 = 43	01 000 = 8	1 0000 = 16	0000 0000 0000 0100= 4

The instruction would be:

sw 16, 4(8) # store word from register 16 to address [8] + 4

Response-time Metric

- Maximizing performance means minimizing response (execution) time

$$\text{Performance} = \frac{1}{\text{Execution time}}$$

- Performance of Processor P_1 is better than P_2 is, for a given work load L, P_1 takes less time to execute L than P_2 does

$$\text{Performance}(P_1) > \text{Performance}(P_2) \text{ w.r.t } L$$

$$\Rightarrow \text{Exection time}(P_1, L) < \text{Execution time}(P_2, L)$$

- Relative performance capture the performance ratio of of Processor P_1 compared to P_2 is, for the same work load

$$\frac{\text{CPU Performance}(P_2)}{\text{CPU Performance}(P_1)} = \frac{\text{Totel execution time}(P_1)}{\text{Totel execution time}(P_2)}$$

CPU Time (Cont.)

- CPU execution time can be measured by running the program
- The clock cycle is usually published by the manufacturer
- Measuring the CPI and instruction count is not trivial
- Instruction counts can be measured by: a software profiling, using an architecture simulator, using hardware counters on some architecture
- The CPI depends on many factors including: processor structure, memory system, the mix of instruction types and the implementation of these instructions
- Designers sometimes uses the following formula:

$$\text{CPU clock cycles} = \sum_{i=1}^{n} CPI_i \times C_i$$

Where: C_i is the count of number of instructions of class *i* executed
CPI_i is the average number of cycles per instruction for that instruction class
n is the number of different instruction classes

Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

	Instruction_count	CPI	Clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	
Processor organization		X	X
Technology			X

* Slide is courtesy of Mary Jane Irwin

An Example

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			Σ = 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?

  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster

- What if two ALU instructions could be executed at once?

  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

* Slide is courtesy of Mary Jane Irwin

Consider two different implementations, M1 and M2, of the same instruction set. There are four classes of instructions (A, B, C, and D) in the instruction set.

M1 has a clock rate of 500 MHz while M2's clock rate is 750 MHz. The average number of cycles for each instruction class of M1 and M2 are shown in the following table:

Class	CPI for this class on M1	CPI for this class on M2
A	1	2
B	2	2
C	3	4
D	4	4

A) Assume that peak performance is defined as the fastest rate that a machine can execute an instruction sequence chosen to maximize that rate. What are the peak performances of M1 and M2 as instructions per second?

B) If the number of instructions executed in a certain program is divided equally among the classes of instructions:
   1. How much faster is M2 than M1?
   2. At what clock rate would M1 have the same performance as the 750-MHz version of M2?

A) If we choose all the instructions from category with the lowest CPI, it will give us the peak performance. By that said, for M1 we will just use Class A, and for M2 we can use Class A and B.

Peak Performance for M1 (Instructions per second):

CPI = 1 → one instruction per cycle.

Clock Rate = 500MHZ → clock time = 1/500MHZ = 2ns

Instructions	Seconds	→x = 1/2ns = 500* 10^6 Instructions
1	2ns	
x	1	

Peak Performance for M2 (Instructions per second):

CPI = 2 → one instruction per two cycles.

Clock Rate = 750MHZ → clock time = 1/750MHZ = $\frac{100}{75} ns$ = 1.3ns

Instructions	Seconds	→x =$375 * 10^6$ Instructions
1	$2*\frac{100}{75} ns$	
x	1	

B)

(1) Assume we have *N* instructions.

Number of cycles on M1: $\frac{1}{4}N * 1 + \frac{1}{4}N * 2 + \frac{1}{4}N * 3 + \frac{1}{4}N * 4 = 2.5N$

Number of cycles on M2: $\frac{1}{4}N * 2 + \frac{1}{4}N * 2 + \frac{1}{4}N * 4 + \frac{1}{4}N * 4 = 3N$

Execution time on M1 = Number of Cycles * Cycle time = 2.5N * 2ns = 5N ns

Execution time on M2 = Number of Cycles * Cycle time = 3N * 1.3ns =3.9N ns

5N/3.9N = 1.29 times M2 is faster than M1

(2) Execution time on M1 = Execution time on M2

2.5N * x = 3.9N ns → x = 1.56ns (clock time)

clock rate =1/clock time = 641 MHZ

Using MIPS as a Performance Metric

- MIPS stands for Million Instructions Per Second and is one of the simplest metrics, which is valid in a limited context

$$\text{MIPS (native MIPS)} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

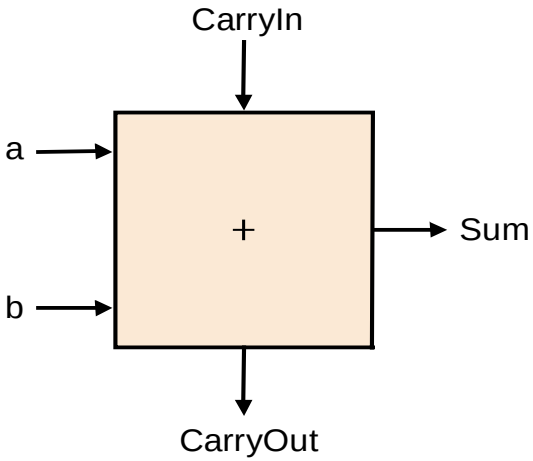
- There are three problems with MIPS:
  - MIPS specifies the instruction execution rate but does not take into account the capabilities of the instructions
  - Computers does not have the same MIPS rating, as MIPS varies between programs on the same computer
  - MIPS can vary inversely with performance (see next example)

The use of MIPS is simple and intuitive, faster machines have bigger MIPS

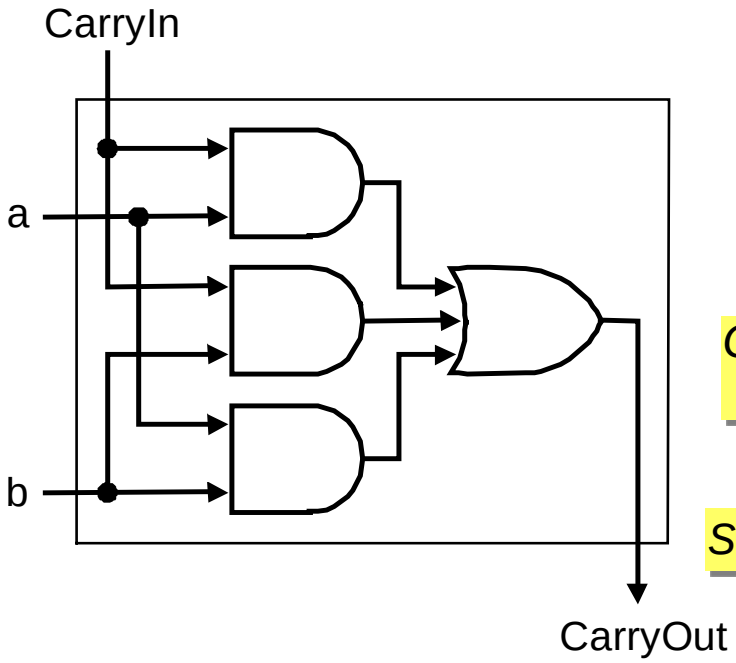
Review Question

- There are times when we want to add a collection of numbers together. Suppose you want to add four 4-bit numbers A, B, E and F using 1-bit full adders. Assume that the time delay through one gate is T and inverters do not introduce delays. Calculate the time for adding four 4-bit numbers using such organizations while having AND and OR gates with two input signals

A 1-Bit Arithmetic Unit



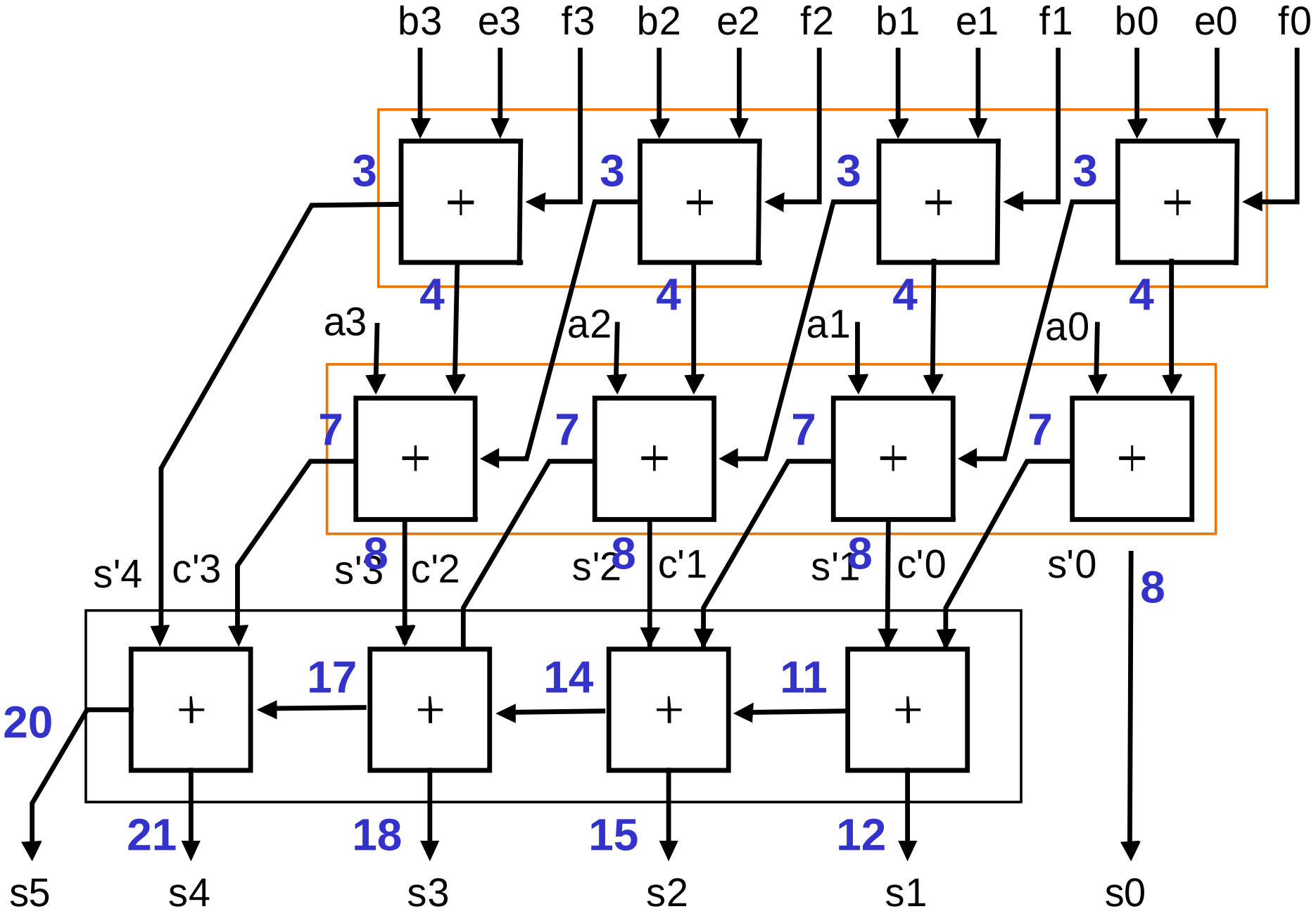
Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



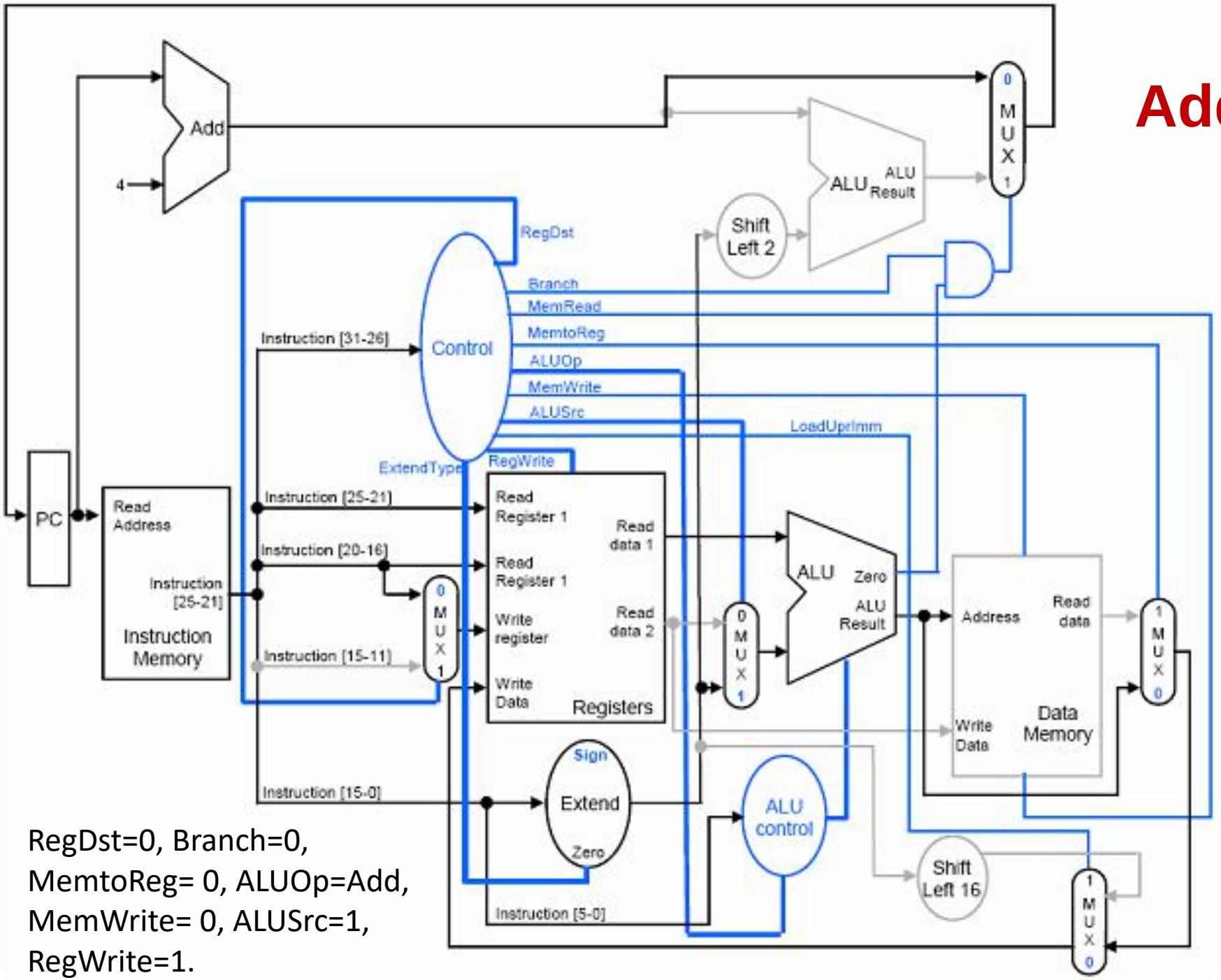
A single bit adder has 3 inputs, two operands and a carry-in and generates a sum bit and a carry-out to be passed to the next 1-bit adder

$$CarryOut = (b.CarryIn) + (a.CarryIn) + (a.b) + (a.b.CarryIn)$$
$$= (b.CarryIn) + (a.CarryIn) + (a.b)$$

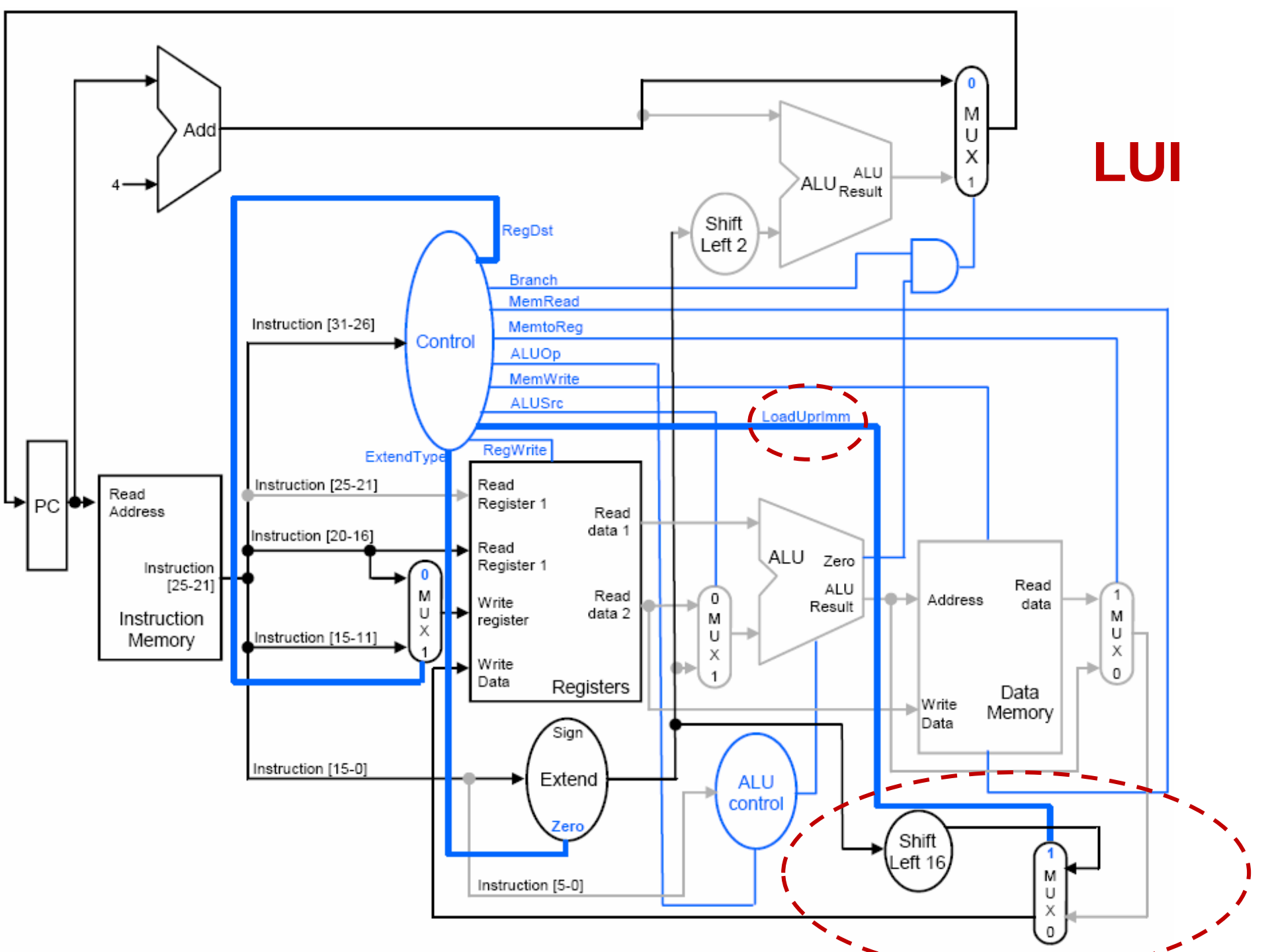
$$Sum = (a.\overline{b}.\overline{CarryIn}) + (\overline{a}.b.\overline{CarryIn}) + (\overline{a}.\overline{b}.CarryIn) + (a.b.CarryIn)$$


b3 e3 f3 b2 e2 f2 b1 e1 f1 b0 e0 f0
3 3 3 3
4 4 4 4
a3 a2 a1 a0
7 7 7 7
8 8 8
s'4 c'3 s'3 c'2 s'2 c'1 s'1 c'0 s'0
8
20 17 14 11
21 18 15 12
s5 s4 s3 s2 s1 s0

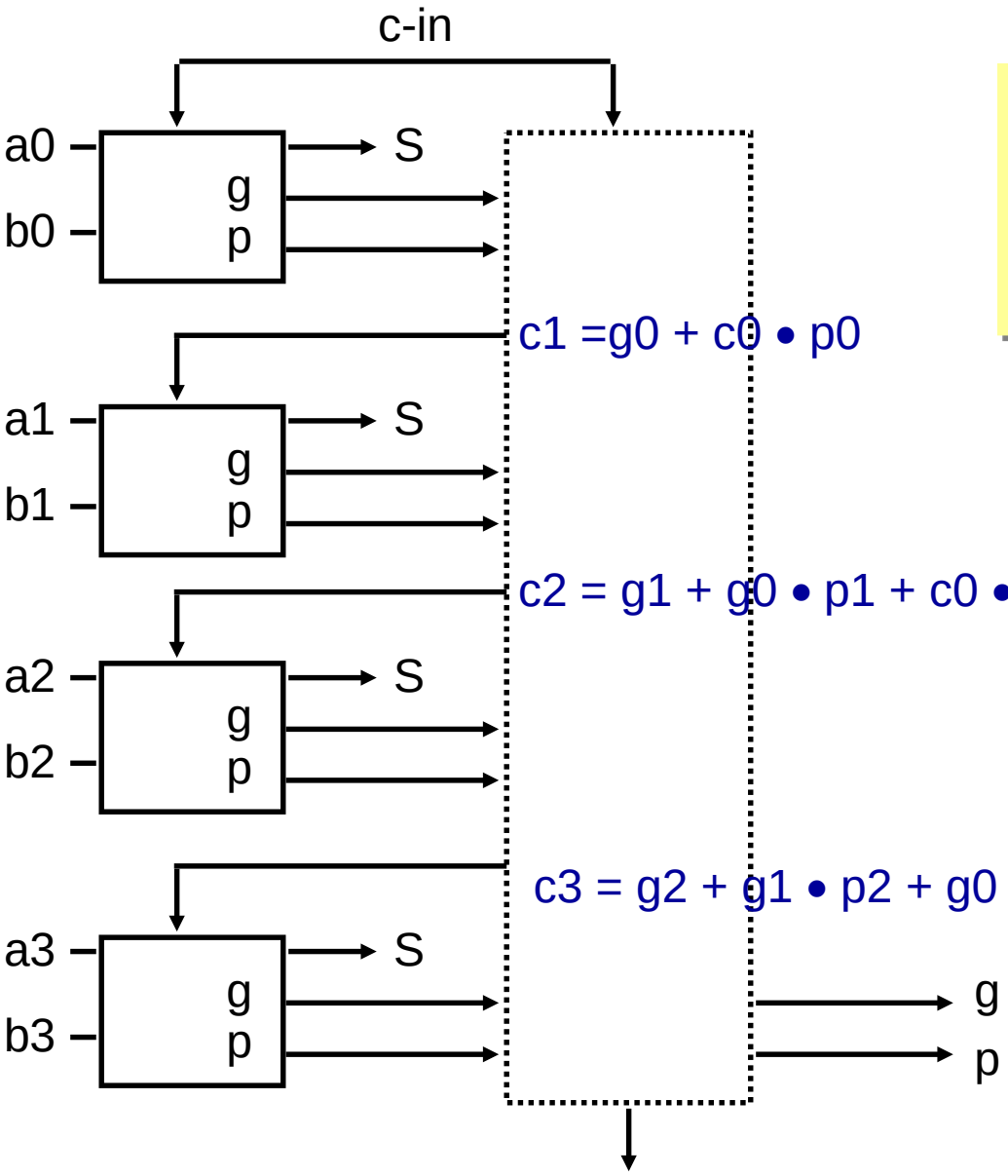
Addi



RegDst=0, Branch=0,
MemtoReg= 0, ALUOp=Add,
MemWrite= 0, ALUSrc=1,
RegWrite=1.


LUI
Add
4
0
M
U
X
1
ALU
ALU Result
Shift Left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
Control
Instruction [31-26]
LoadUprimm
ExtendType
RegWrite
PC
Read Address
Instruction [25-21]
Instruction Memory
Instruction [25-21]
Instruction [20-16]
Instruction [15-11]
Instruction [15-0]
Instruction [5-0]
Read Register 1
Read Register 1
Write register
Write Data
Read data 1
Read data 2
Registers
Zero
ALU Result
Address
Read data
Write Data
Data Memory
Sign
Extend
Zero
ALU control
Shift Left 16

Carry Lookahead (propagate & generate)



$$c_{i+1} = (b_i \,.\, c_i) + (a_i \,.\, c_i) + (a_i \,.\, b_i)$$
$$= (a_i \,.\, b_i) + c_i \,.\, (a_i + b_i)$$
$$= g_i + c_i \,.\, p_i$$

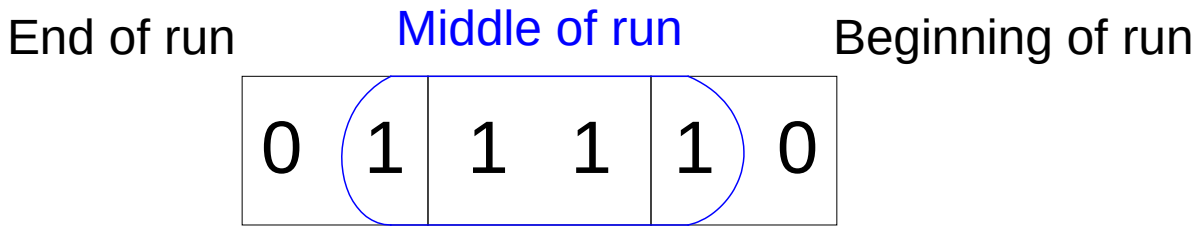
a	b	c-out	
0	0	0	“kill”
0	1	c-in	“propagate”
1	0	c-in	“propagate”
1	1	1	“generate”

p = a or b
g = a and b

* Figure is courtesy of Dave Patterson

Booth's Algorithm

Current bit	Bit to the right	Explanation	Example
1	0	Beginning of a run of 1s	00001111000
1	1	Middle of a run of 1s	00001111000
0	1	End of a run of 1s	00001111000
0	0	Middle of a run of 0s	00001111000



❶ Depending on the current and previous bits, do one of the following

00: Middle of a string of 0s ⇒ no arithmetic operation

01: End of a string of 1s ⇒ add the multiplicand to the left half of the product

10: Beginning of a string of 1s ⇒ subtract multiplicand from left half of the product

11: Middle of a string of 1s ⇒ no arithmetic operation

❷ Shift the Product register to the right for 1 bit

Booth's algorithm works for both signed and unsigned numbers

Example (unsigned numbers)

Compare the multiplication algorithm (version 3) and Booth's algorithm applied to getting the product of 2 × 6 using only 4-bit binary representation

Multiplicand	Original Algorithm		Booth's Algorithm	
	Step	Product	Step	Product
0010	Initial value	0000 0110	Initial value	0000 0110 0
0010	1a: 0 ⇒ no operation	0000 0110	1a: 00 ⇒ no operation	0000 0110 0
0010	2: Shift right Product	0000 0011	2: Shift right Product	0000 0011 0
0010	1a: 1 ⇒ Prod = Prod + Mcand	0010 0011	1a: 10 ⇒ Prod = Prod - Mcand	1110 0011 0
0010	2: Shift right Product	0001 0001	2: Shift right Product	1111 0001 1
0010	1a: 1 ⇒ Prod = Prod + Mcand	0011 0001	1a: 11 ⇒ no operation	1111 0001 1
0010	2: Shift right Product	0001 1000	2: Shift right Product	1111 1000 1
0010	1a: 0 ⇒ no operation	0001 1000	1a: 01 ⇒ Prod = Prod + Mcand	0001 1000 1
0010	2: Shift right Product	0000 1100	2: Shift right Product	0000 1100 0

- Booth's algorithm uses both the current bit and the previous bit to determine its course of action
- Extend the sign when shifting to preserve the sign (*arithmetic right shift*)

Example (signed numbers)

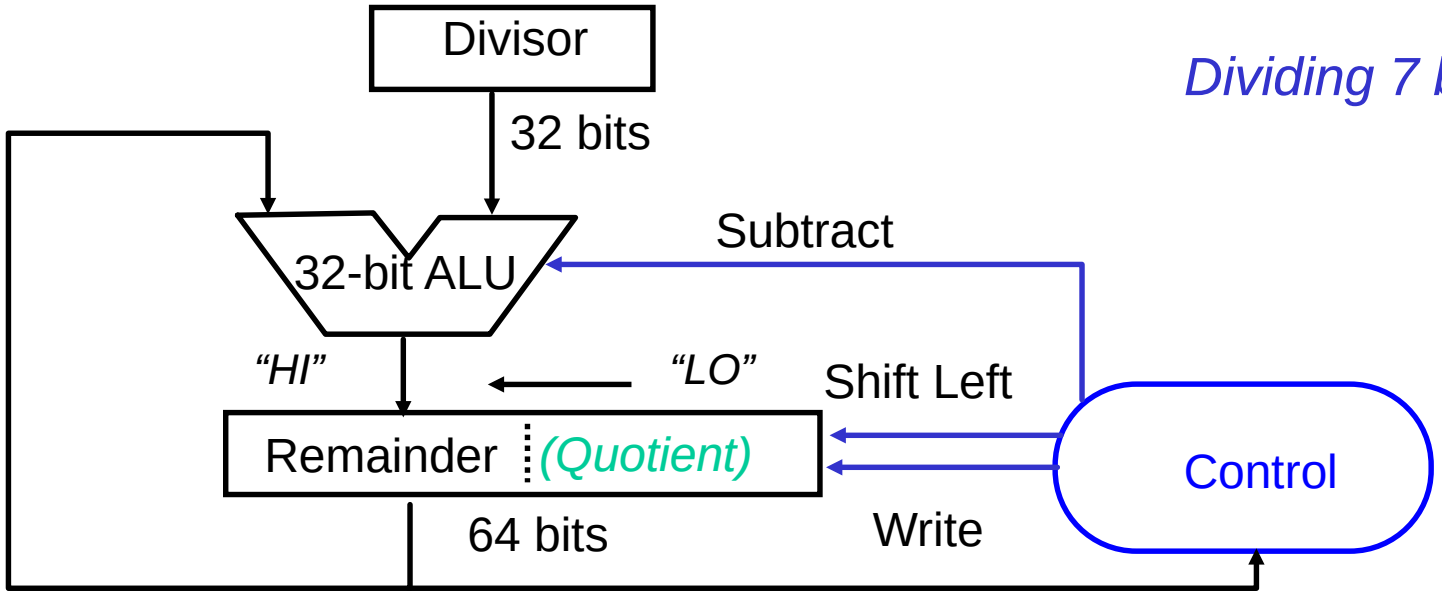
Follow Booth's algorithm to get the product of 2 × -3 using only 4-bit binary representation

Iteration	Step	Multiplicand	Product
0	Initial value	0010	0000 1101 0
1	1a: 10 ⇒ Prod = Prod - Mcand	0010	1110 1101 0
	2: Shift right Product	0010	1111 0110 1
2	1a: 01 ⇒ Prod = Prod + Mcand	0010	0001 0110 1
	2: Shift right Product	0010	0000 1011 0
3	1a: 10 ⇒ Prod = Prod - Mcand	0010	1110 1011 0
	2: Shift right Product	0010	1111 0101 1
4	1a: 11 ⇒ no operation	0010	1111 0101 1
	2: Shift right Product	0010	1111 1010 1

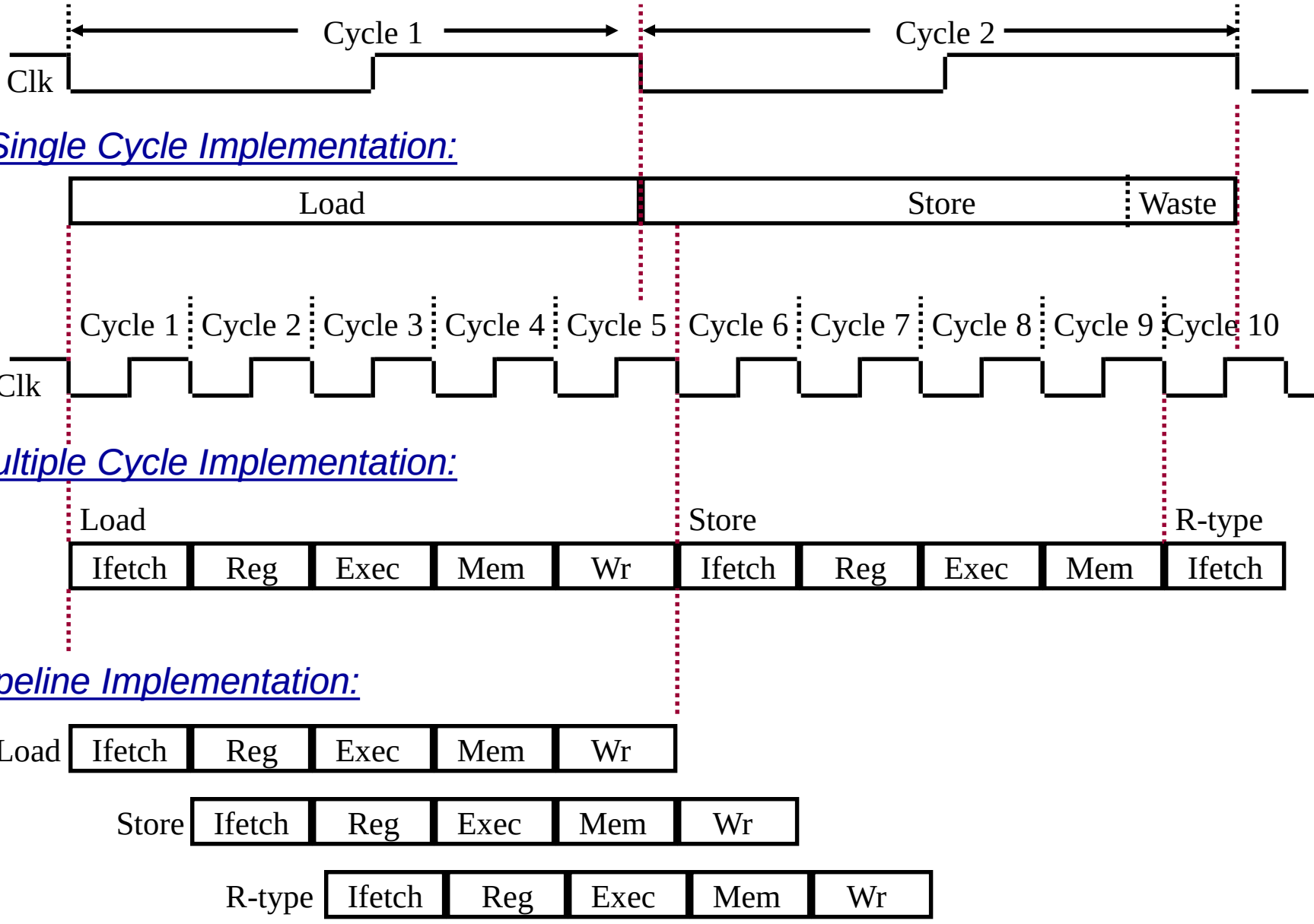
Divide Hardware

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift Rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 ⇒ +Div, shift left R, R0=0	0010	0001 1100
2	2: Rem = Rem - Div	0010	1111 1100
	3b: Rem < 0 ⇒ +Div, shift left R, R0=0	0010	0011 1000
3	2: Rem = Rem - Div	0010	0001 1000
	3a: Rem ≥ 0 ⇒ shift left R, R0=1	0010	0011 0001
4	2: Rem = Rem - Div	0010	0001 0001
	3a: Rem ≥ 0 ⇒ shift left R, R0=1	0010	0010 0011
	Shift left half of Rem right 1	0010	0001 0011

Dividing 7 by 2

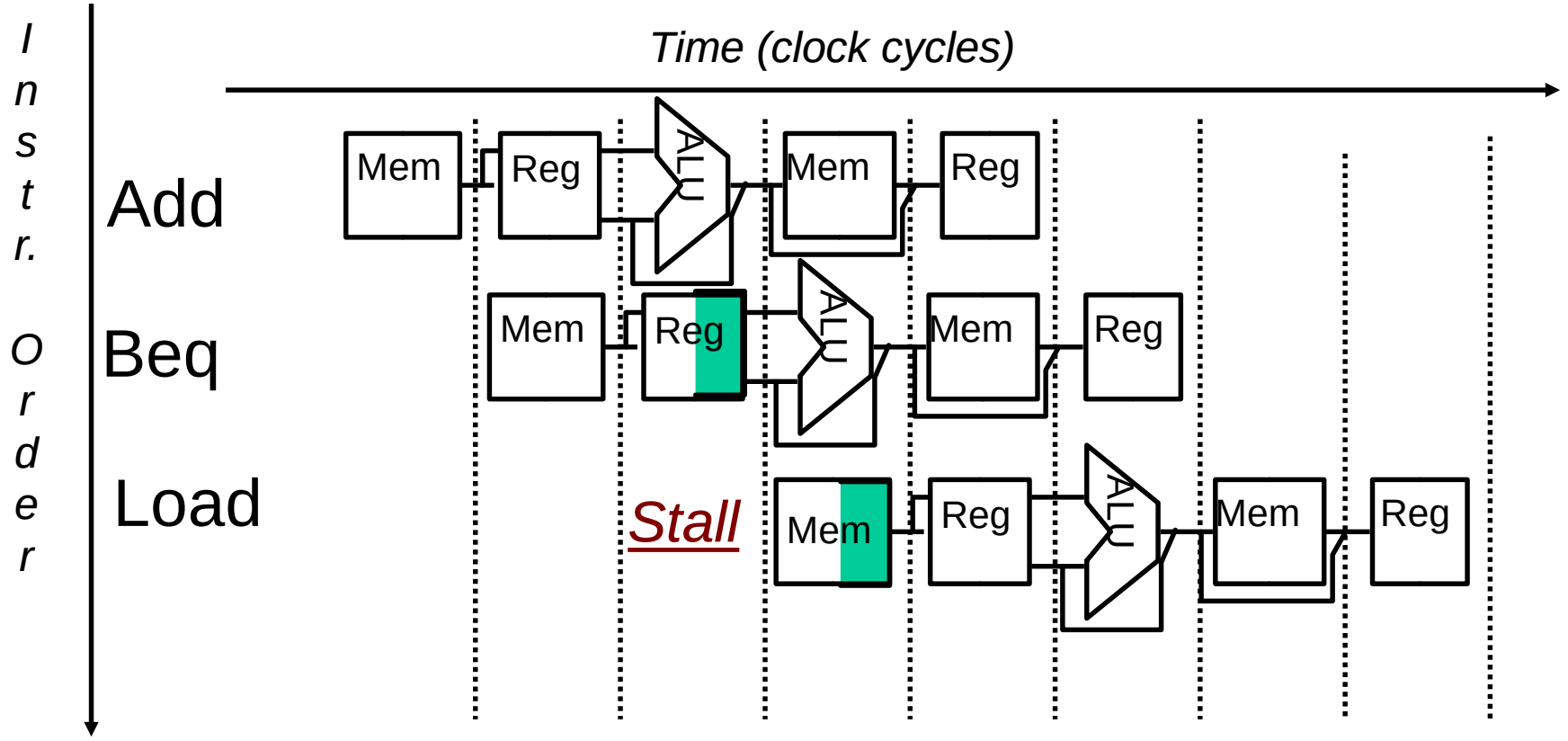


Single Cycle, Multiple Cycle, vs. Pipeline



Control Hazard

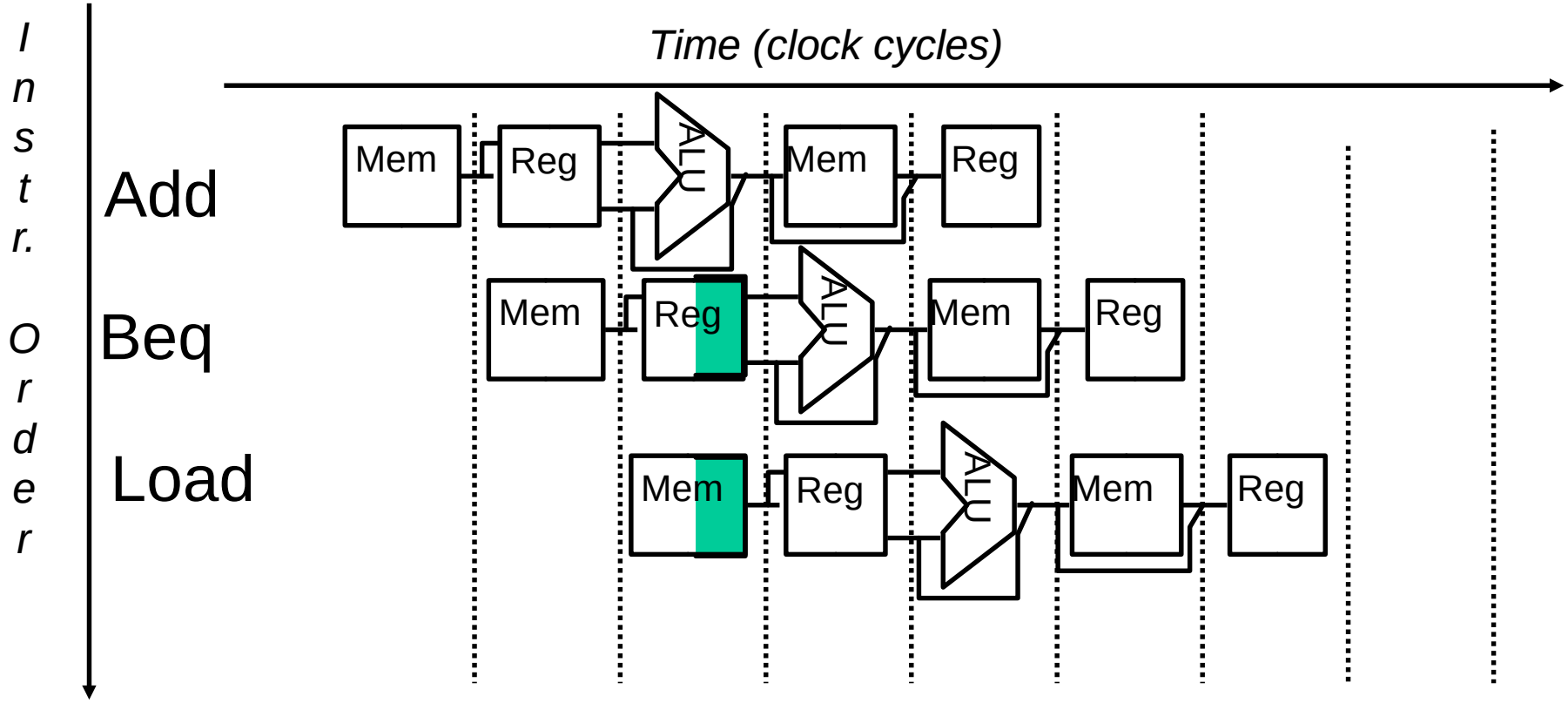
- Stall: wait until decision is clear
  - It is possible to move up decision to 2nd stage by adding hardware to check registers as being read



❑ Impact: 2 clock cycles per branch instruction ⇒ slow

Control Hazard Solution

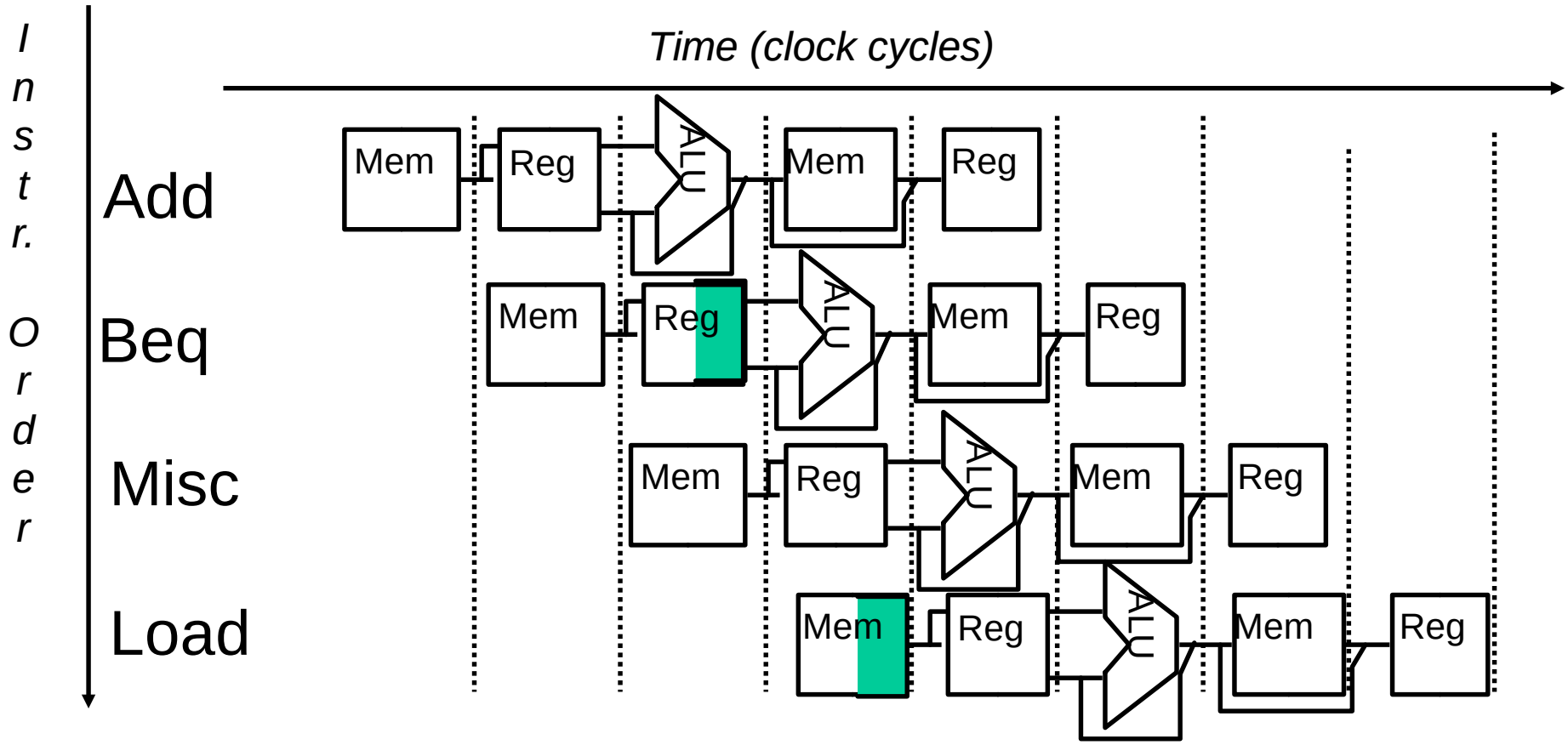
- Predict: guess one direction then back up if wrong
  - Predict not taken



- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right 50% of time)
- More dynamic scheme: history of 1 branch (90%)

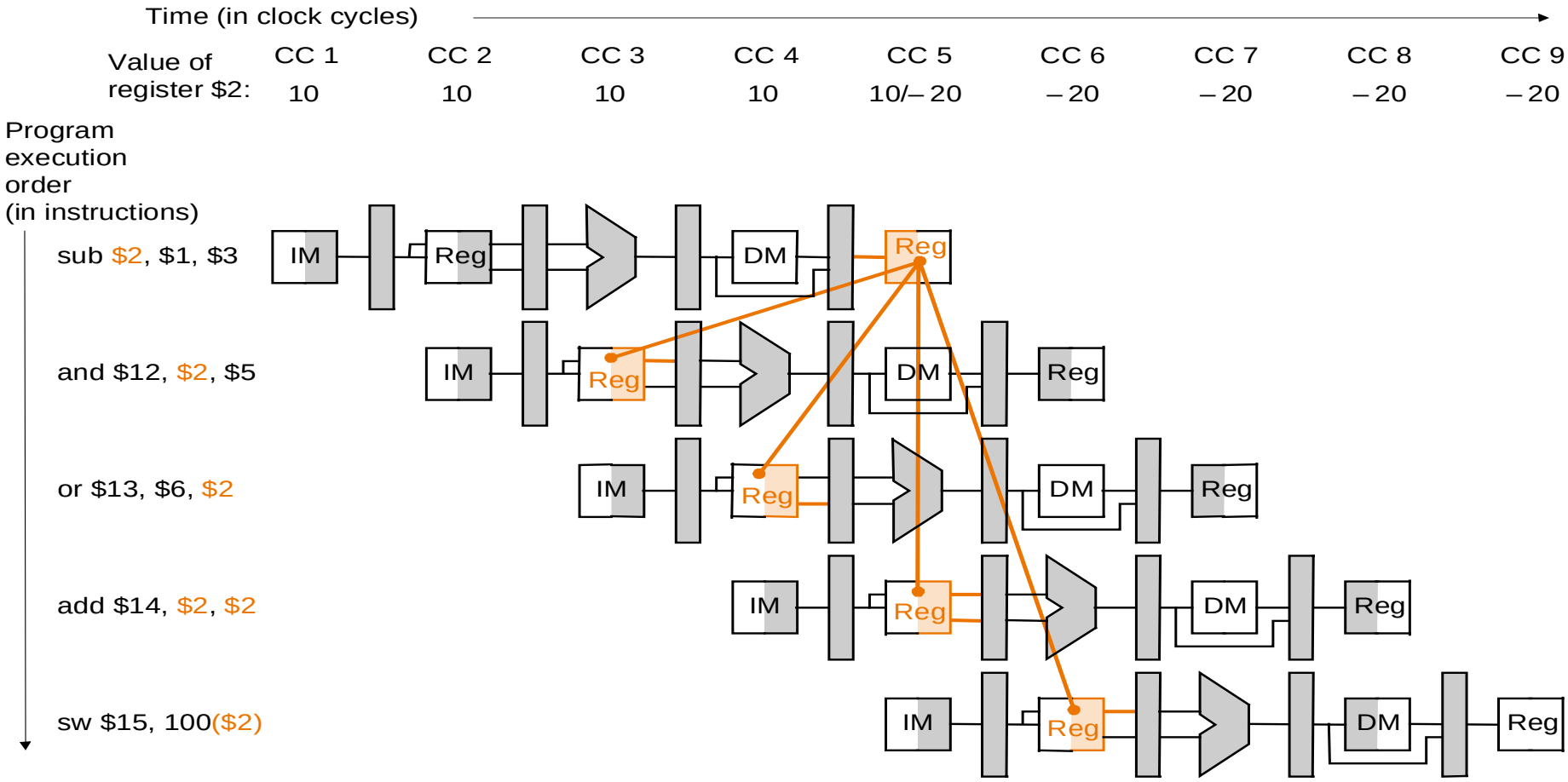
Control Hazard Solution

- Redefine branch behavior (takes place after next instruction) "delayed branch"



❑ Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (50% of time)

Data Hazards

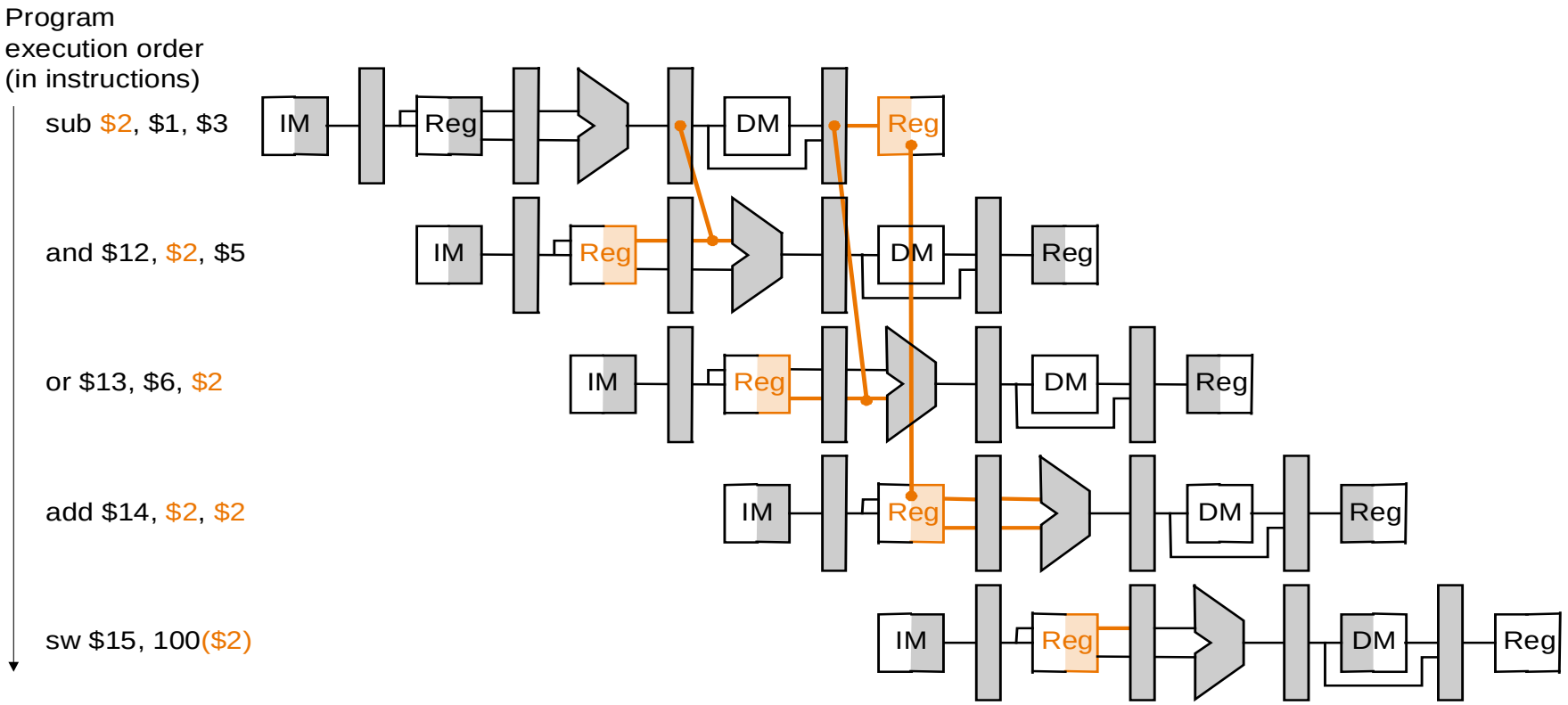


- Generally caused by data dependence among instructions
- Can cause erroneous semantics if went undetected and avoided
- Raised when an instruction depends on result of a prior instruction still in the pipeline and attempts to use item before it is ready

Data Forwarding

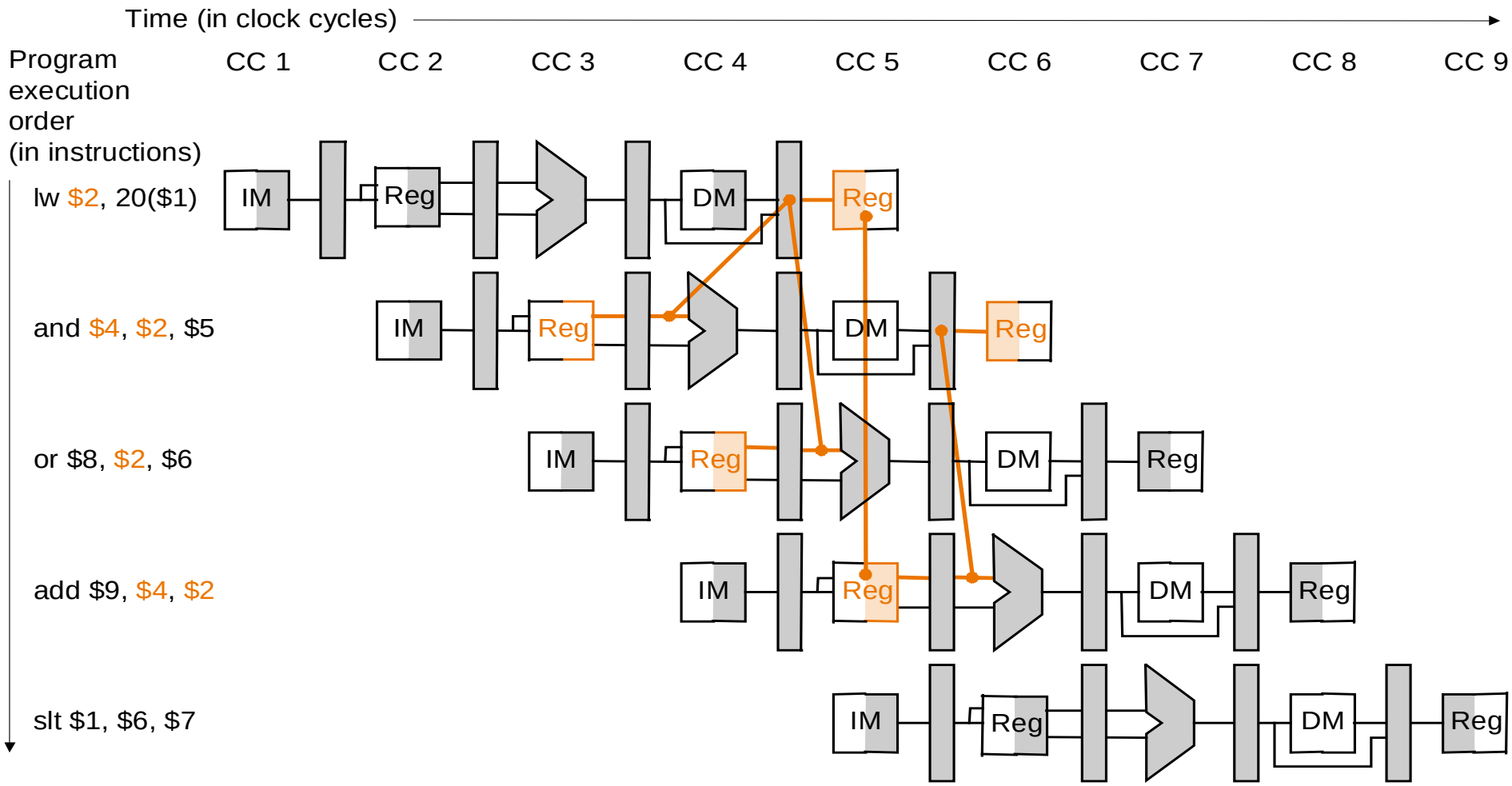
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



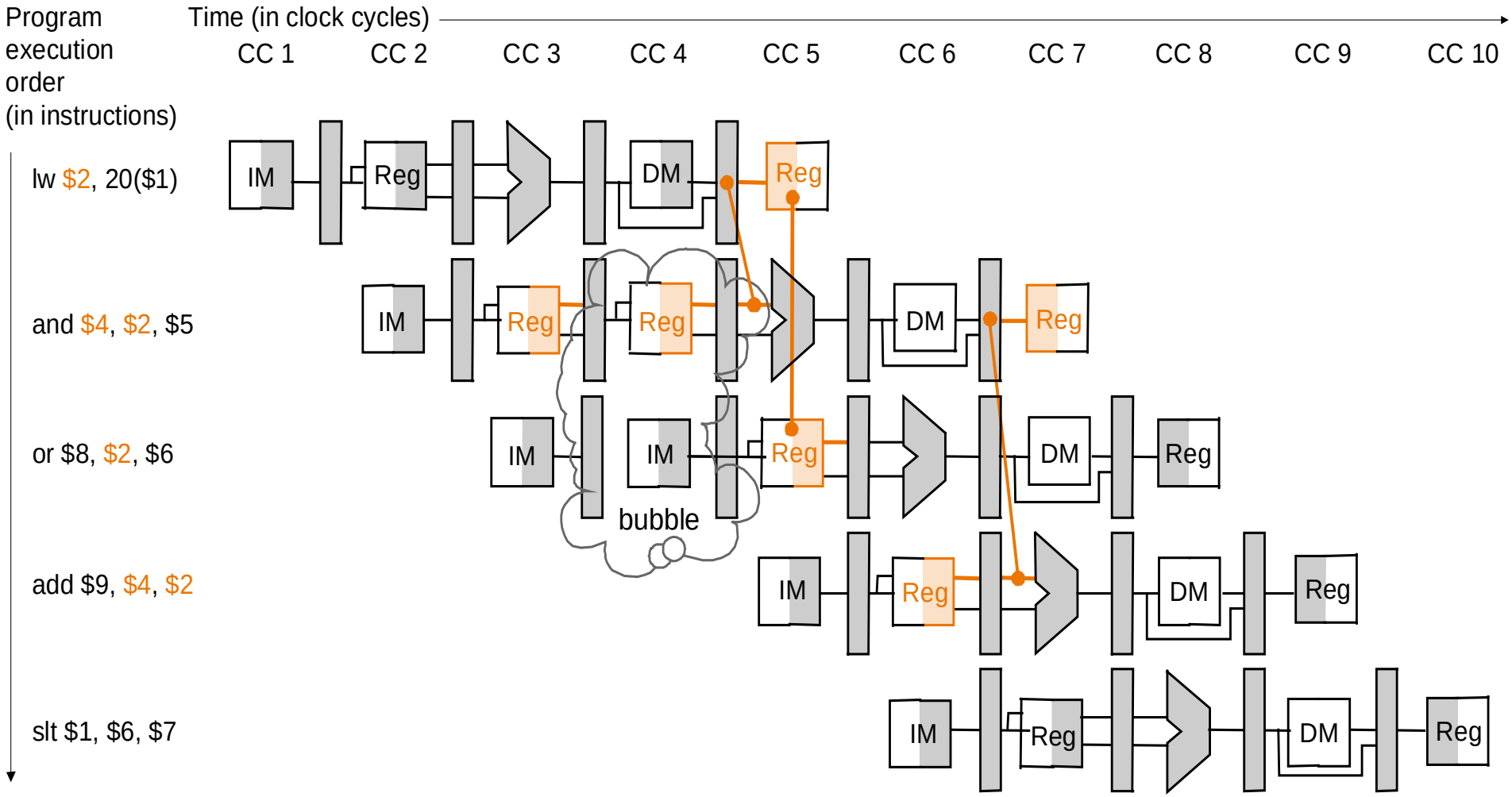
- Detecting data hazard conditions allows using intermediate results
- Forward only when there is a WB stage (check the RegWrite signal)

Data Hazard Caused by Load



- Dependencies backward in time are hazards
- Cannot solve with forwarding
- Must delay/stall instruction dependent on loads

Hazard Detection Unit



Solution: need to stall the pipeline

Example

For the following sequence of instructions:

```
LW   $1,  40($6)
ADD  $6,  $2,   $2
SW   $6,  50($1)
```

A) Assume there is no forwarding in this pipeline processor, indicate hazards and add *NOP* instructions to eliminate them.

	1	2	3	4	5	6	7	8	9
LW $1, 40($6)	IF	ID	EX	ME	WB				
ADD $6, $2, $2		IF	ID	EX	ME	WB			
NOP			NOP	NOP	NOP	NOP	NOP		
NOP				NOP	NOP	NOP	NOP	NOP	
SW $6,50($1)					IF	ID	EX	ME	WB

B) Repeat (A) while assuming full forwarding.

	1	2	3	4	5	6	7
LW $1, 40($6)	IF	ID	EX	ME	WB		
ADD $6, $2, $2		IF	ID	EX	ME	WB	
SW $6,50($1)			IF	ID	EX	ME	WB

Example (Cont.)

For the following sequence of instructions:

```
LW   $1,  40($6)
ADD  $6,  $2,   $2
SW   $6,  50($1)
```

C) Add NOP instructions to this code to eliminate hazards if there is ALU-ALU forwarding only (no forwarding from the MEM to the EX stage)?

.

	1	2	3	4	5	6	7	8
LW $1, 40($6)	IF	ID	EX	ME	WB			
ADD $6, $2, $2		IF	ID	EX	ME	WB		
NOP			NOP	NOP	NOP	NOP	NOP	
SW $6,50($1)				IF	ID	EX	ME	WB

Example

Assume that the following MIPS code is executed on a pipelined processor with a five-stage pipeline, full forwarding, and a predict-taken branch predictor:

```
            add     $1, $5, $3
Label:      sw      $1, 0($2)
            add     $2, $2, $3
            beq     $2, $4, Label          # Not taken
            add     $5, $5 ,$1
            sw      $1, 0($2)
```

A) Draw the pipeline execution diagram for this code, assuming that branches execute in the EX stage and there is misprediction (i.e., else clause is pursued).

Instructions	Cycles											
	1	2	3	4	5	6	7	8	9	10	11	12
add $1, $5, $3	IF	ID	EX	ME	WB							
Label: sw $1, 0($2)		IF	ID	EX	ME	WB						
add $2, $2, $3			IF	ID	EX	ME	WB					
beq $2, $4, Label				IF	ID	EX	ME	WB				
Label: sw $1, 0($2)					IF	ID						
add $2, $2, $3						IF						
add $5, $5 ,$1							IF	ID	EX	ME	WB	
sw $1, 0($2)								IF	ID	EX	ME	WB

Example (Cont.)

As pointed out in class, the effect of control hazards can be reduced by moving branch execution into the ID stage. This approach involves a dedicated comparator in the ID stage. For the given code, what is the speed-up achieved by moving branch execution into the ID stage? Still here is misprediction (i.e., else clause is pursued).

Instructions	Cycles											
	1	2	3	4	5	6	7	8	9	10	11	12
add $1, $5, $3	IF	ID	EX	ME	WB							
Label: sw $1, 0($2)		IF	ID	EX	ME	WB						
add $2, $2, $3			IF	ID	EX	ME	WB					
beq $2, $4, Label				IF	ID	EX	ME	WB				
Label: sw $1, 0($2)					IF							
add $5, $5 ,$1						IF	ID	EX	ME	WB		
sw $1, 0($2)							IF	ID	EX	ME	WB	

Example

For this problem, assume that all branches are resolved in ID stage and are perfectly predicted (this eliminates all control hazards). For the following fragment of MIPS code:

```
        LW  $5, -16($5)
        SW   $4, -16($4)
        LW   $3, -20($4)
        BEQ $2, $0, Label    ;Assume $2 ≠ $0
        ADD $1, $5, $4
Label:  SUB $2, $1, $3
```

If we only have one memory (for both instruction and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data.

•What is the total execution time of this instruction sequence in the 5-stage pipeline that only has one memory?

Question 2.A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LW $5, -16($5)	F	D	X	M	W										
SW $4, -16($4)		F	D	X	M	W									
LW $3, -20($4)			F	D	X	M	W								
BEQ $2, $0, Label ;Assume $2 != $0				*S*	*S*	*S*	F	D							
ADD $1, $5, $4								F	D	X	M	W			
Label: SUB $2, $1, $3									F	D	X	M	W		

Example

For following code, assume that the loop index ($10) is a multiple of 8:

```
Loop:   LW     $2,    0($10)
        SUB    $4,    $2,    $3
        SW     $4,    0($10)
        LW     $5,    4($10)
        SUB    $6,    $5,    $3
        SW     $6,    4($10)
        ADDI   $10,   $10,   8
        BNE    $10,   $30,   Loop
```

Schedule this code (reorder the instructions and make any necessary changes) for fast execution on the 5-stage MIPS pipeline. Assume data forwarding and not-taken prediction of conditional branching.

Baseline

Question 3	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Loop: LW $2, 0($10)	F	D	X	M	W										
SUB $4, $2, $3		F	D	*S*	X	M	W								
SW $4, 0($10)			F	*S*	D	X	M	W							
LW $5, 4($10)					F	D	X	M	W						
SUB $6, $5, $3						F	D	*S*	X	M	W				
SW $6, 4($10)							F	*S*	D	X	M	W			
ADDI $10, $10, 8									F	D	X	M	W		
BNE $10, $30, Loop										F	D	X			

Example (Cont.)

For following code, assume that the loop index ($10) is a multiple of 8:

```
Loop:   LW    $2,   0($10)
        SUB   $4,   $2,   $3
        SW    $4,   0($10)
        LW    $5,   4($10)
        SUB   $6,   $5,   $3
        SW    $6,   4($10)
        ADDI  $10,  $10,  8
        BNE   $10,  $30,  Loop
```

Schedule this code (reorder the instructions and make any necessary changes) for fast execution on the 5-stage MIPS pipeline. Assume data forwarding and not-taken prediction of conditional branching.

Optimized

Question 3	1	2	3	4	5	6	7	8	9	10	11	12
Loop: LW $2, 0($10)	F	D	X	M	W							
LW $5, 4($10)		F	D	X	M	W						
SUB $4, $2, $3			F	D	X	M	W					
SW $4, 0($10)				F	D	X	M	W				
SUB $6, $5, $3				F	D	X	M	W				
ADDI $10, $10, 8					F	D	X	M	W			
SW $6, -4($10)						F	D	X	M	W		
BNE $10, $30, Loop							F	D	X			