TCS 802
Advanced Computer Architecture

Pipelined Processing & Superscalar Techniques

- Linear & Nonlinear pipelines
- Instruction Pipelines & Arithmetic Operations
- Superscalar and super-pipeline

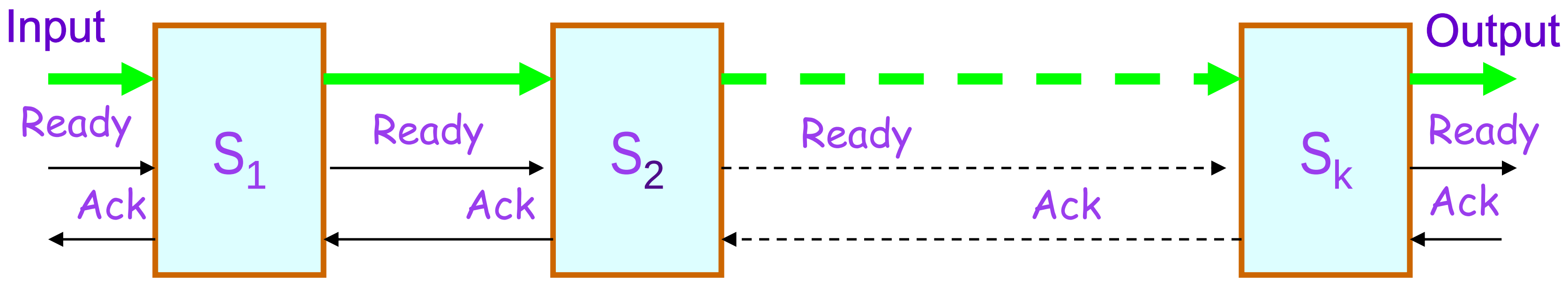
Principles of Pipelining

- A pipeline may be compared directly with an assembly line in a manufacturing plant. Thus
  - Input task or process is subdivided into a sequence of subtasks;
  - Each subtask is executed by a specialized hardware stage;
  - Many such hardware stages operate concurrently;
  - When successive tasks are streamed into the pipeline they are executed in an overlapped fashion at the subtask level.
  - The creation of the correct sequence of subtasks is crucial to the performance of the pipeline.
  - Slowest subtask is the bottleneck in the pipeline.

Linear Pipeline

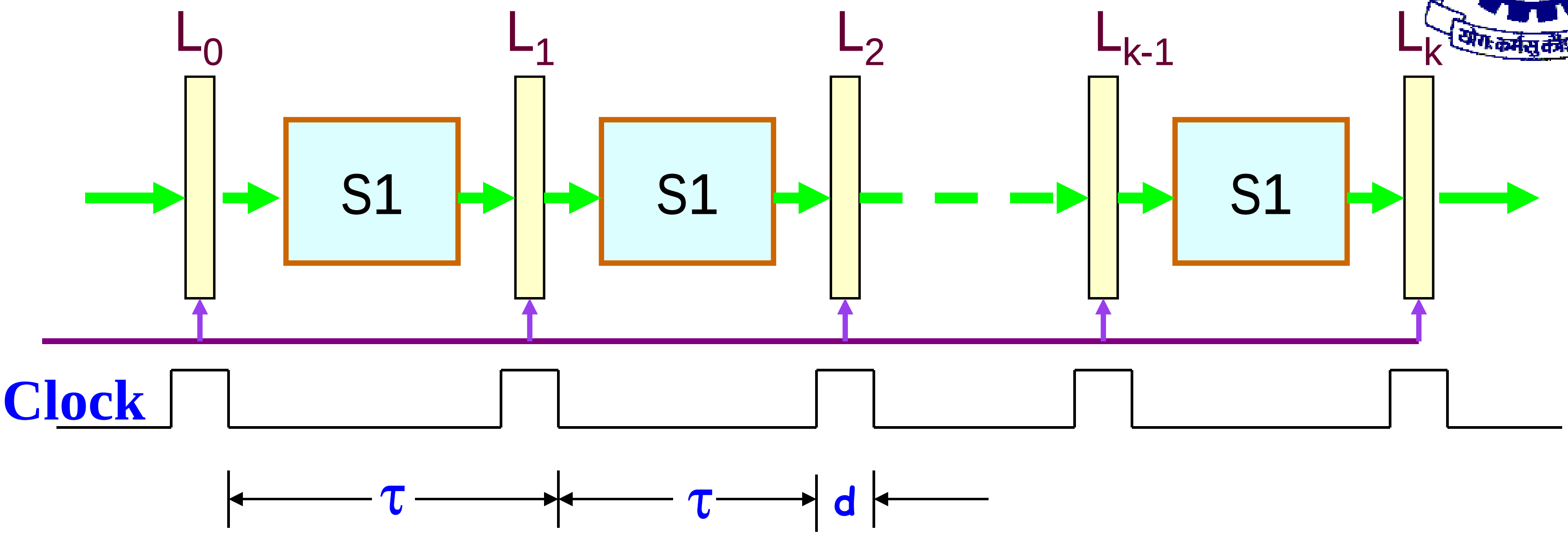
- Processing Stages are linearly connected
- Perform fixed function
- Synchronous Pipeline
  - Clocked latches between Stage i and Stage i+1
  - Equal delays in all stages
- Asynchronous Pipeline (Handshaking)

Asynchronous Pipeline



- Asynchronous pipelines are useful in designing communication channels in massage passing multicomputer
- Asynchronous pipelines may have variable throughput rate. Different amount of delay may be experienced in different stages.

Synchronous Pipeline



Reservation Table

Time

S1	X			
S2		X		
S3			X	
S4				X

- Reservation Table of a four stage linear pipeline

Time

	T_1	T	T_3	T	T_5	T	T_7	T
S1	X	2 X	X	4 X	X	6		8
S2		X	X	X	X	X		
S3			X	X	X	X	X	
S4				X	X	X	X	X

- 5 tasks on 4 stages

Clock period and frequency

- Define

  τ_i to be time delay due to logic circuitry in stage S_i ,
  d to be time delay of each interface latch.

- Then
  - The clock period, τ , of a linear pipeline is given by

    $\tau = \max\{\tau_i\} + d$

    $= \tau_M + d$

  - The frequency, $f = 1 / \tau$

    $= 1 / [\tau_M + d]$ (i.e., reciprocal of clock period)

Speedup, Efficiency & Throughput

Speedup

- Ideally, a linear pipeline of k stages can process n task in k + (n-1) clock so total time required is

$$T_K = \{ k + (n-1)\} \tau$$

- Time required for nonpipelined processing

$$T_1 = n\ k\ \tau$$

Speedup factor $S_K = T_1 / T_k$

$$S_K = n\ k / k + (n-1)$$

Speedup, Efficiency & Throughput

- Pipeline efficiency, $\eta = S_k / k$

  $= n / k + (n-1)$

  So efficiency = 1 when $n \rightarrow \infty$ and

  lower bound of η is $1 / k$ when $n=1$

- Throughput i.e. Number of task performed per unit time

  $H_k = n / \{ k + (n-1)\} \tau = n f / k +(n-1)$

Non Linear Pipelines

- Variable functions
- Feed-Forward
- Feedback

Linear Instruction Pipelines

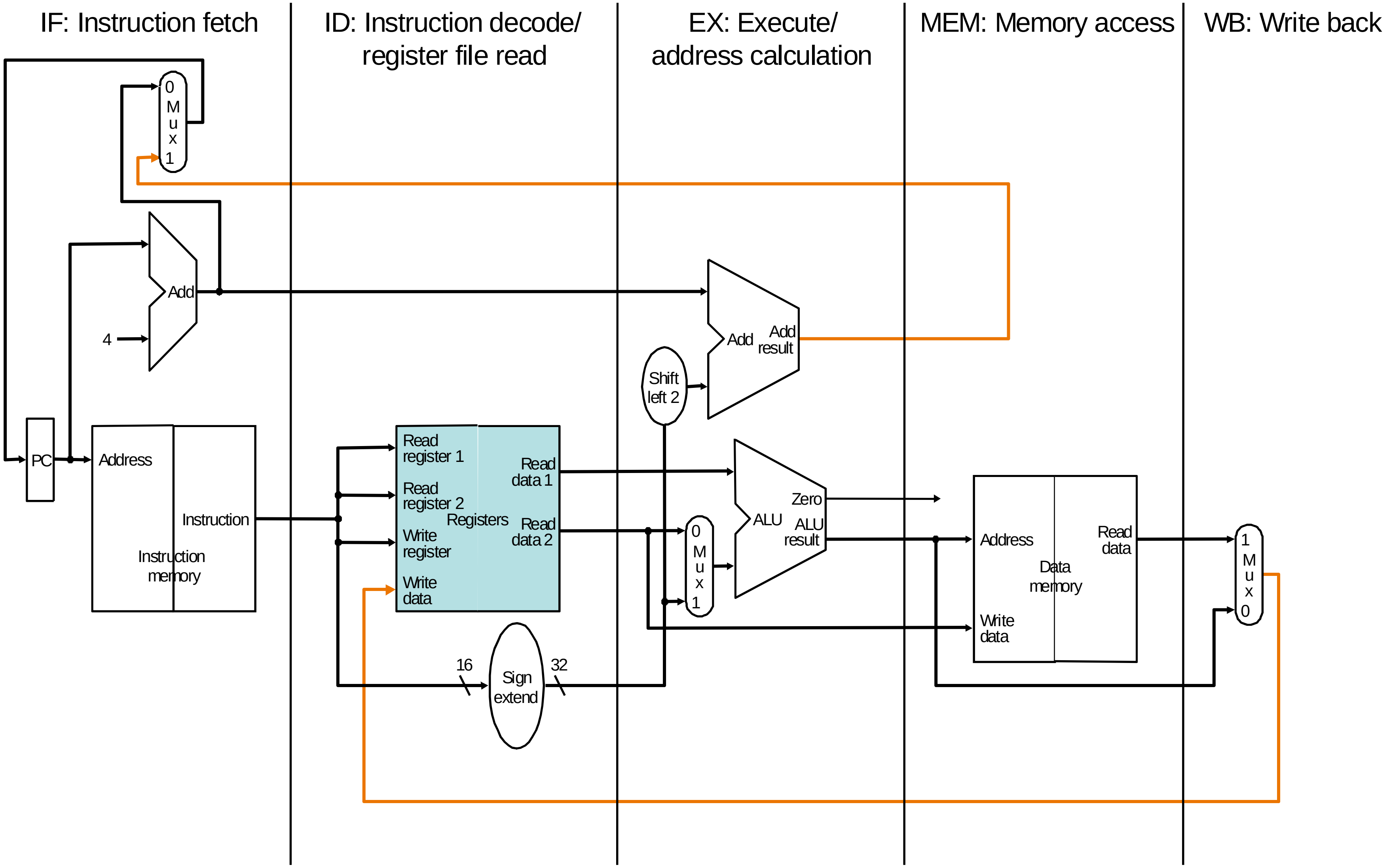
- Assume the following instruction execution phases:
  - Fetch (F)
  - Decode (D)
  - Operand Fetch (O)
  - Execute (E)
  - Write results (W)

Execution

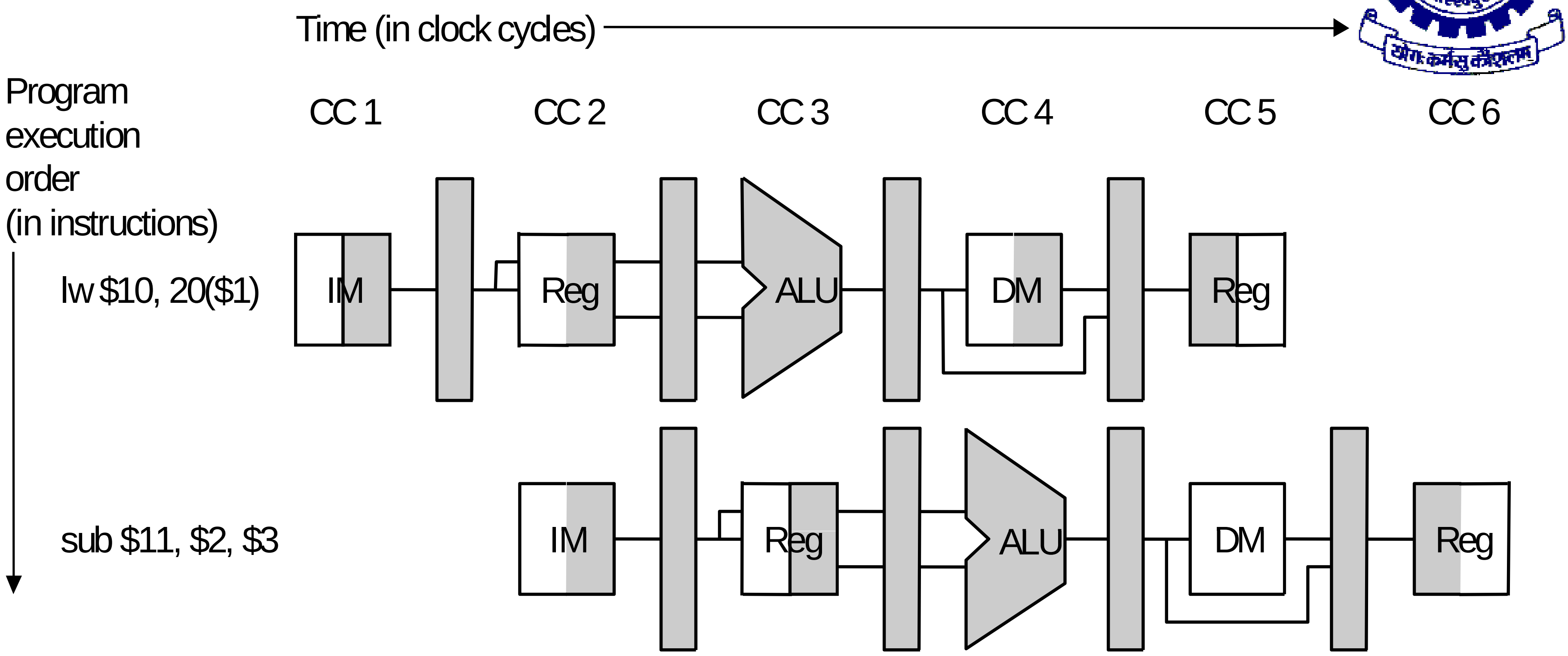
F	I_1	I_2	I_3					
D		I_1	I_2	I_3				
O			I_1	I_2	I_3			
E				I_1	I_2	I_3		
W					I_1	I_2	I_3	

Basic idea: start from single cycle impl.

What do we need to add to actually split the datapath into stages?



Graphically Representing Pipelines



Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6

Program
execution
order
(in instructions)

lw $10, 20($1)
IM Reg ALU DM Reg

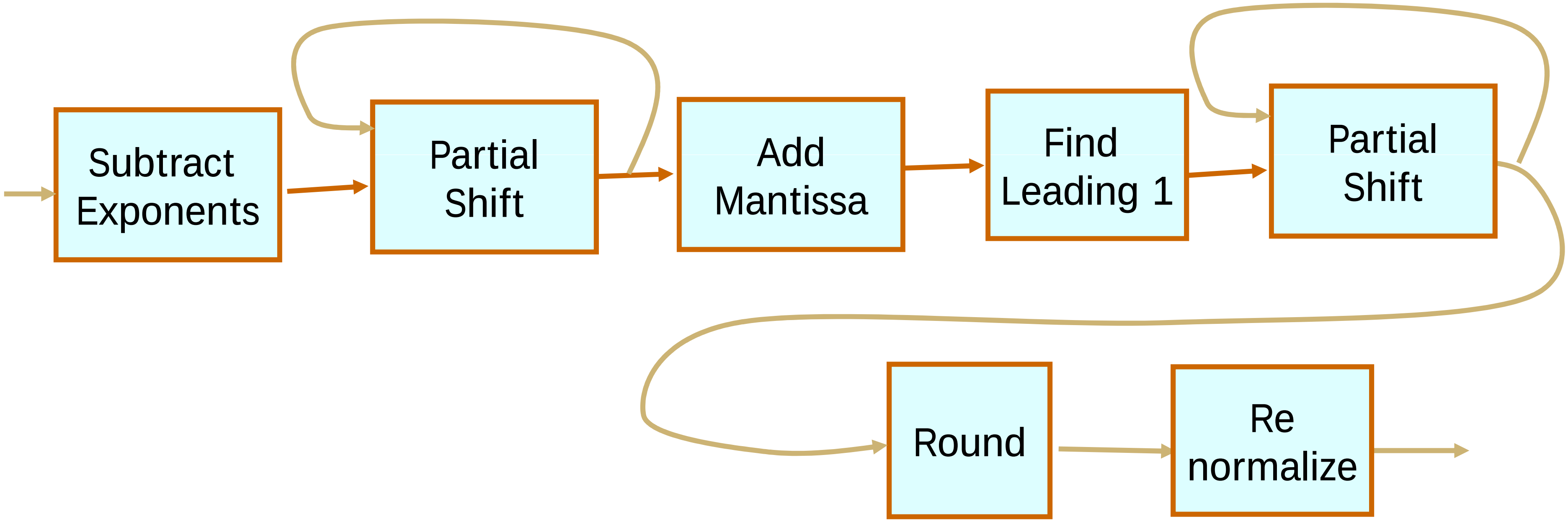
sub $11, $2, $3
IM Reg ALU DM Reg

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Floating Point Multiplication

- Inputs (Mantissa$_1$, Exponenet$_1$), (Mantissa$_2$, Exponent$_2$)
- Add the two exponents → Exponent-out
- Multiple the 2 mantissas
- Normalize mantissa and adjust exponent
- Round the product mantissa to a single length mantissa. You may adjust the exponent

Linear Pipeline for floating-point Addition

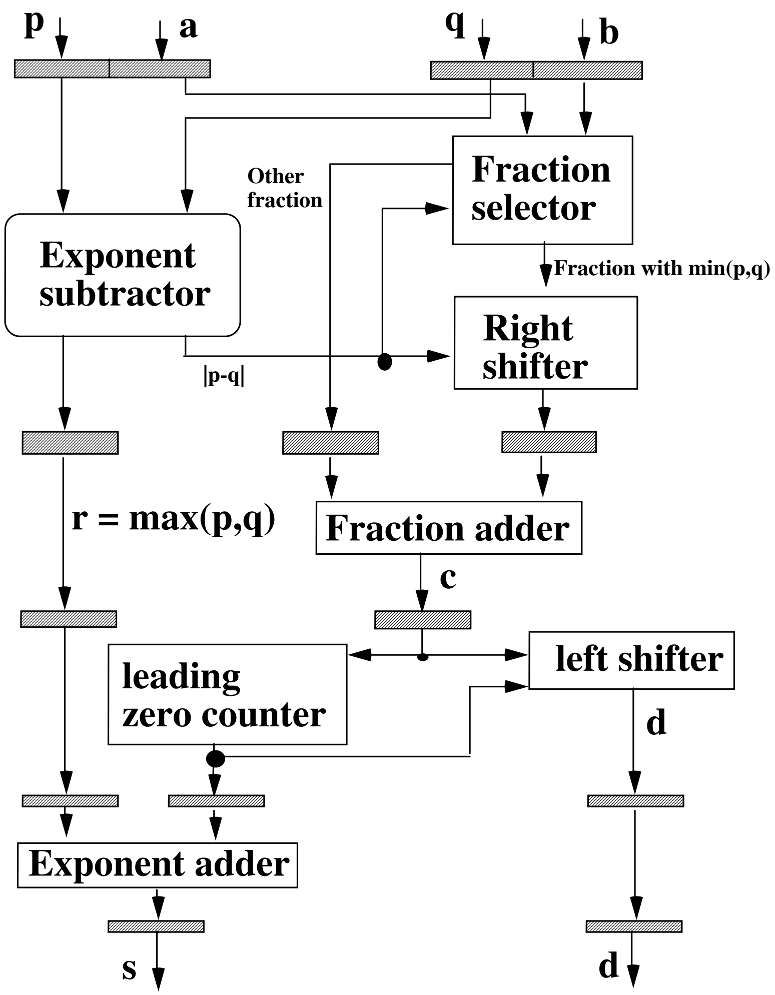


Example

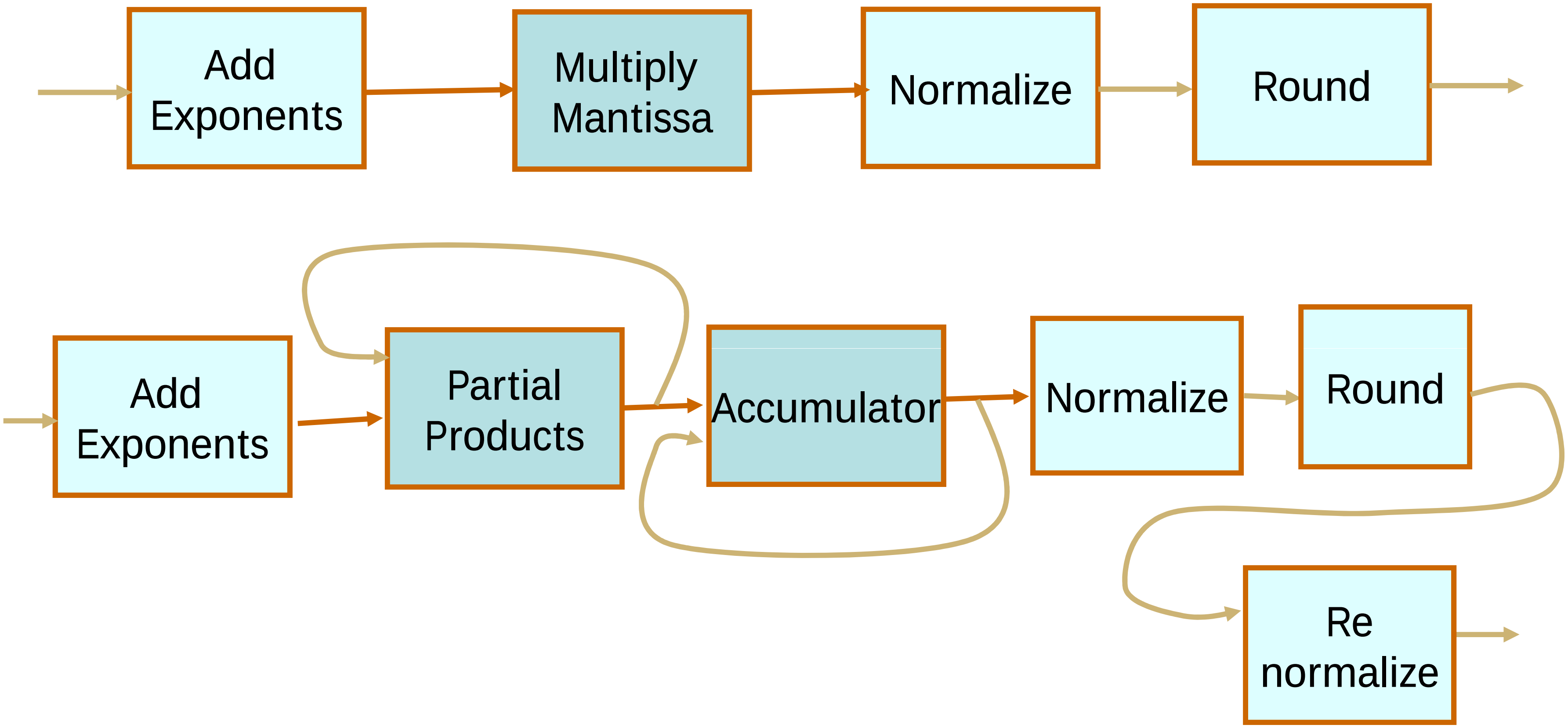
- Linear pipeline with four functional stages. Inputs are two normalised floating-point numbers

  a*2^p and b*2^q

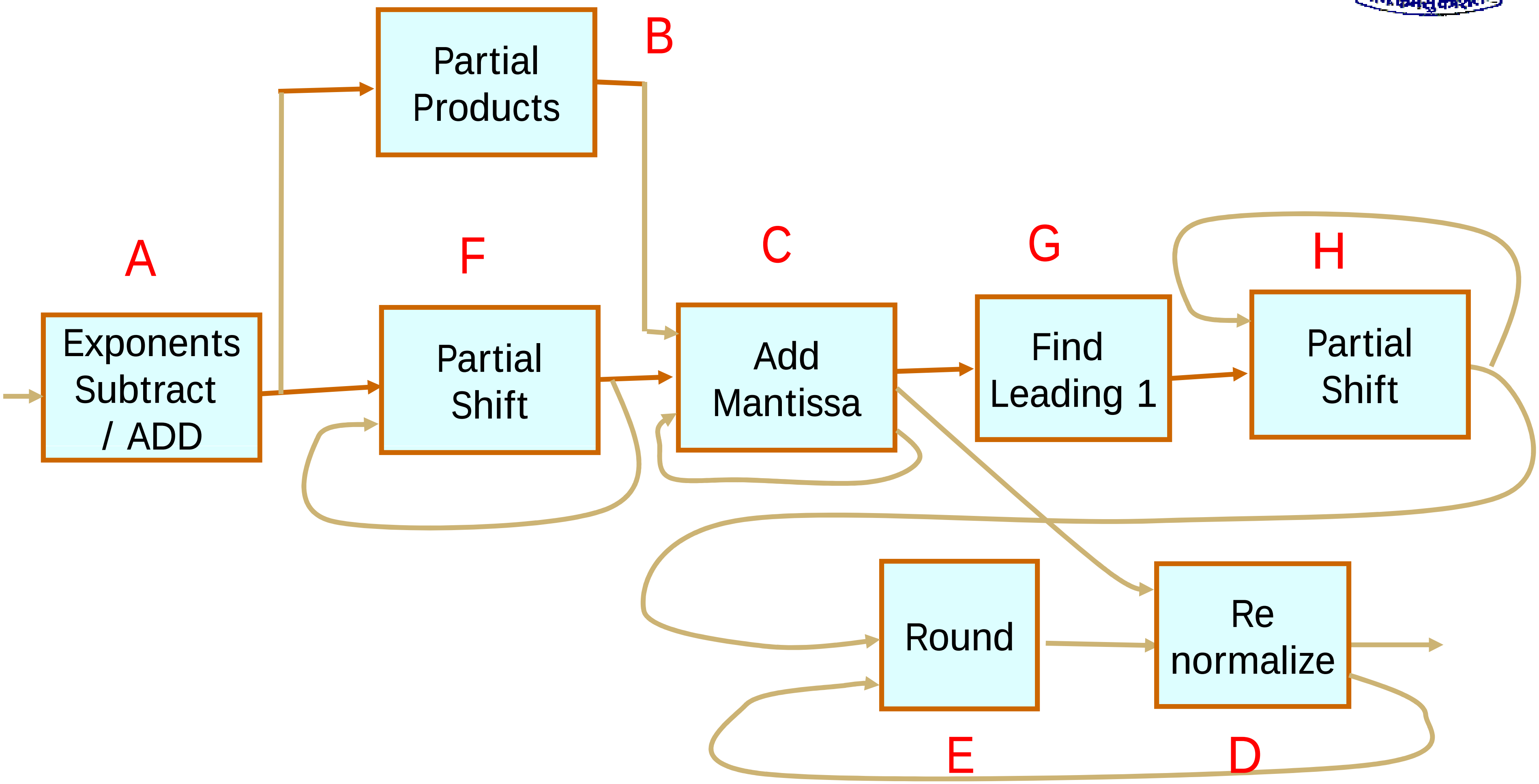
- Output is a normalised floating-point number d* 2^s which is the sum of the two inputs.
- The hardware units other than the latches can all be implemented using combinational logic.
- If time delay of interface latches is 10ns and if the time delays of the four stages are 60, 50, 90 and 80ns, respectively, then cycle time of pipeline can be chosen to be 100ns.



Linear Pipeline for floating-point multiplication



Combined Adder and Multiplier



Reservation Table for Multiply

	1	2	3	4	5	6	7
A	X						
B		X	X				
C			X	X			
D					X		X
E						X	
F							
G							
H							

Reservation Table for Addition

	1	2	3	4	5	6	7	8	9
A	Y								
B									
C				Y					
D									Y
E								Y	
F		Y	Y						
G					Y				
H						Y	Y		

Nonlinear Pipeline Design

- Latency
  - The number of clock cycles between two initiations of a pipeline
- Collision
  - Resource Conflict
- Forbidden Latencies
  - Latencies that cause collisions

Nonlinear Pipeline Design

- Latency Sequence
  - A sequence of permissible latencies between successive task initiations
- Latency Cycle
  - A sequence that repeats the same subsequence
- Collision vector
  - C = (Cm, Cm-1, …, C2, C1), m <= n-1
  - n = number of column in reservation table
  - Ci = 1 if latency i causes collision, 0 otherwise

Mul – Mul Collision (lunch after 1 cycle)

	1	2	3	4	5	6	7
A	X	Z					
B		X	X Z	Z			
C			X	X Z	Z		
D					X	Z	X
E						X	Z
F							
G							
H							

Mul – Mul Collision (lunch after 2 cycle)

	1	2	3	4	5	6	7
A	X		Z				
B		X	X	Z	Z		
C			X	X	Z	Z	
D					X		X Z
E						X	
F							
G							
H							

Mul – Mul Collision (lunch after 3 cycle)

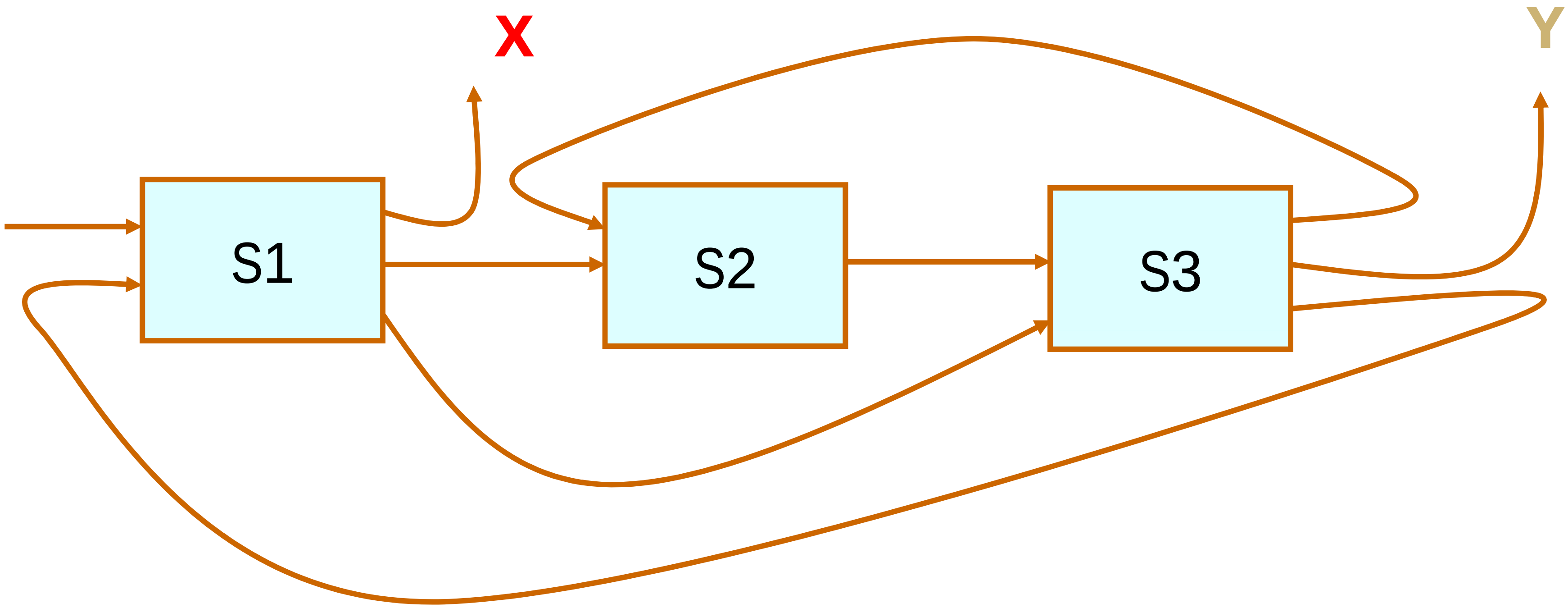
	1	2	3	4	5	6	7
A	X			Z			
B		X	X		Z	Z	
C			X	X		Z	Z
D					X		X
E						X	
F							
G							
H							

Collision Vector for Multiply after Multiply

- Forbidden Latencies: 1, 2
- Collision vector

  0 0 0 0 1 1 → 11
- Maximum forbidden latency = 2 → m = 2

Design Example



Reservation Tables for X & Y

S1	X					X		X
S2		X		X				
S3			X		X		X	

S1	Y				Y	
S2			Y			
S3		Y		Y		Y

Forbidden Latencies

- X after X

2

S1	X1		X2			X1		X2 X1
S2		X1		X2 X1		X2		
S3			X1		X2 X1		X2 X1	

5

S1	X1					X2 X1		X1
S2		X1		X1			X2	
S3			X1		X1		X1	X2

Forbidden Latencies

- X after X

4

S1	X1				X2	X1		X1
S2		X1		X1		X2		X2
S3			X1		X1		X2 X1	

7

S1	X1					X1		X2 X1
S2		X1		X1				
S3			X1		X1		X1	

Collision Vector

- Forbidden Latencies: 2, 4, 5, 7
- Collision Vector =

1 0 1 1 0 1 0

Y after Y

S1	Y		Y		Y	
S2			Y		Y	
S3		Y		Y Y		Y Y

S1	Y				Y Y	
S2			Y			
S3		Y		Y		Y Y

Collision Vector

- Forbidden Latencies: 2, 4
- Collision Vector =

1 0 1 0

Exercise – Find the collision vector

	1	2	3	4	5	6	7
A	X		X	X			
B		X				X	
C					X		X
D				X			

Latency Analysis

- The number of cycles between two initiations is the *latency* between them
- A latency of k → two initiations are separated by k cycles
- Collision → resource conflict between two initiations
- Latencies that cause collision → forbidden latencies

Latency Analysis

- Latency Sequence → a sequence of permissible latencies between successive initiations
- Latency Cycle → a latency sequence that repeats the same subsequence (cycle) indefinitely
- Latency Sequence → 1, 8
- Latencies Cycle → (1,8) → 1, 8, 1, 8, 1, 8 …

Latency Analysis

- Average Latency (of a latency cycle) → sum of all latencies / number of latencies along the cycle
- Constant Cycle → One latency value
- Objective → Obtain the shortest average latency between initiations without causing collisions.

Latency Cycle (1,8)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
X1	X2				X1	X2	X1	X2	X3	X4				X3	X4	X3	X4	X5	X6		
	X1	X2	X1	X2						X3	X4	X3	X4						X5	X6	
		X1	X2	X1	X2	X1	X2				X3	X4	X3	X4	X3	X4				X5	

Average Latency = (1+8)/2 = 4.5

Latency Cycle (6)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
X1					X1	X2	X1				X2	X3	X2				X3	X4	X3		
	X1		X1				X2		X2				X3		X3				X4		
		X1		X1		X1		X2		X2		X2		X3		X3		X3		X4	

Average Latency = 6

Find the collision vector

	1	2	3	4	5	6	7
A	X		X	X			
B		X				X	
C					X		X
D				X			

Collision Vector

- Forbidden Latencies: 1,2, 4
- Collision Vector =

001011

Reservation Tables for X & Y

S1	X					X		X
S2		X		X				
S3			X		X		X	

S1	Y				Y	
S2			Y			
S3		Y		Y		Y

Collision Vector

- Forbidden Latencies: 2, 4, 5, 7
- Collision Vector =

  1 0 1 1 0 1 0
- The next state is obtained by bitwise ORing the initial collision vector with the shifted register
  - C.V. = 1 0 1 1 0 1 0 (first state)

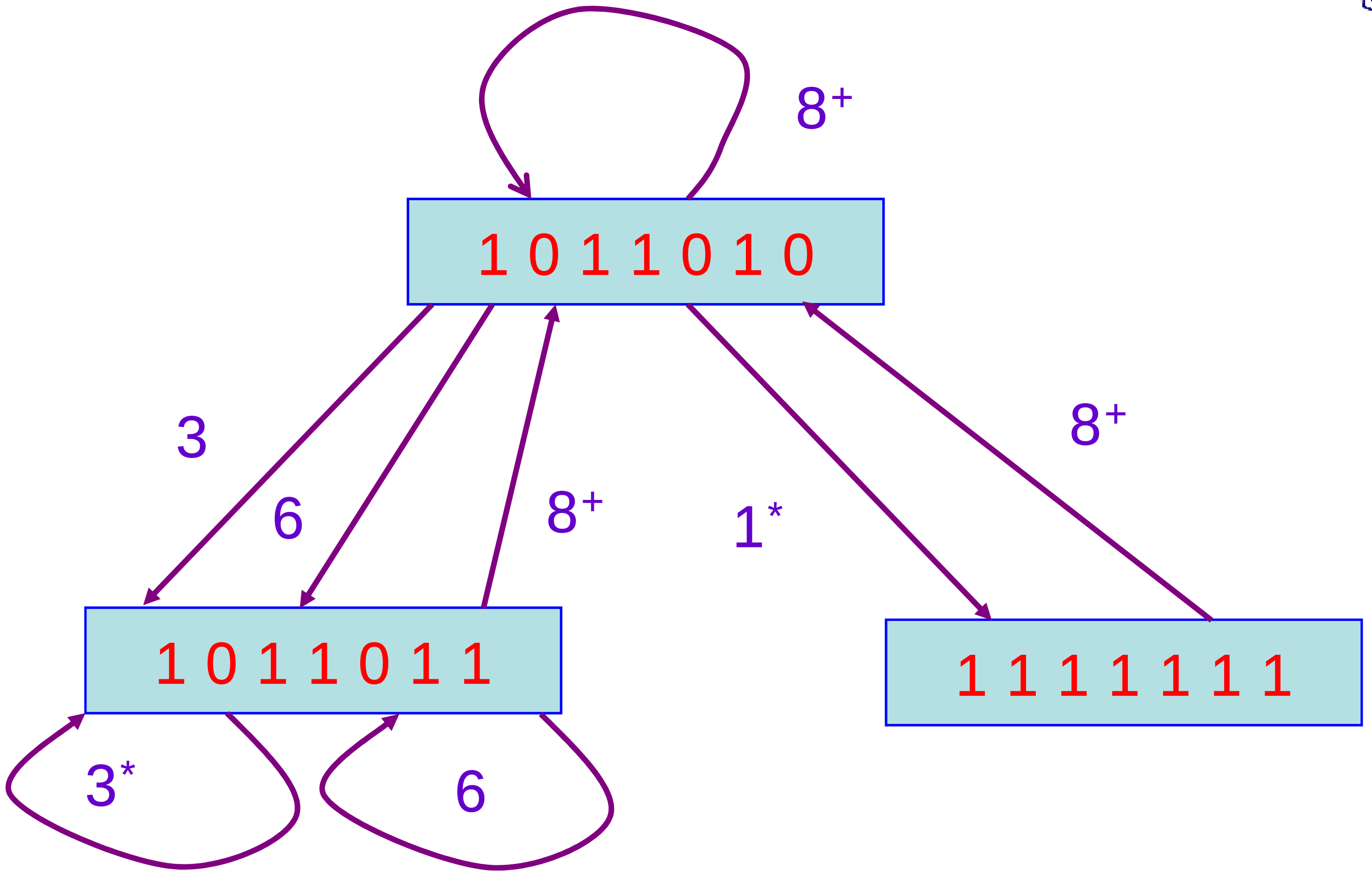
0 1 0 1 1 0 1 C.V. 1-bit right shifted

1 0 1 1 0 1 0 initial C.V.

---------------- OR

1 1 1 1 1 1 1

State Diagram for X after X



Cycles

- Simple cycles → each state appears only once

  (3), (6), (8), (1, 8), (3, 8), and (6,8)

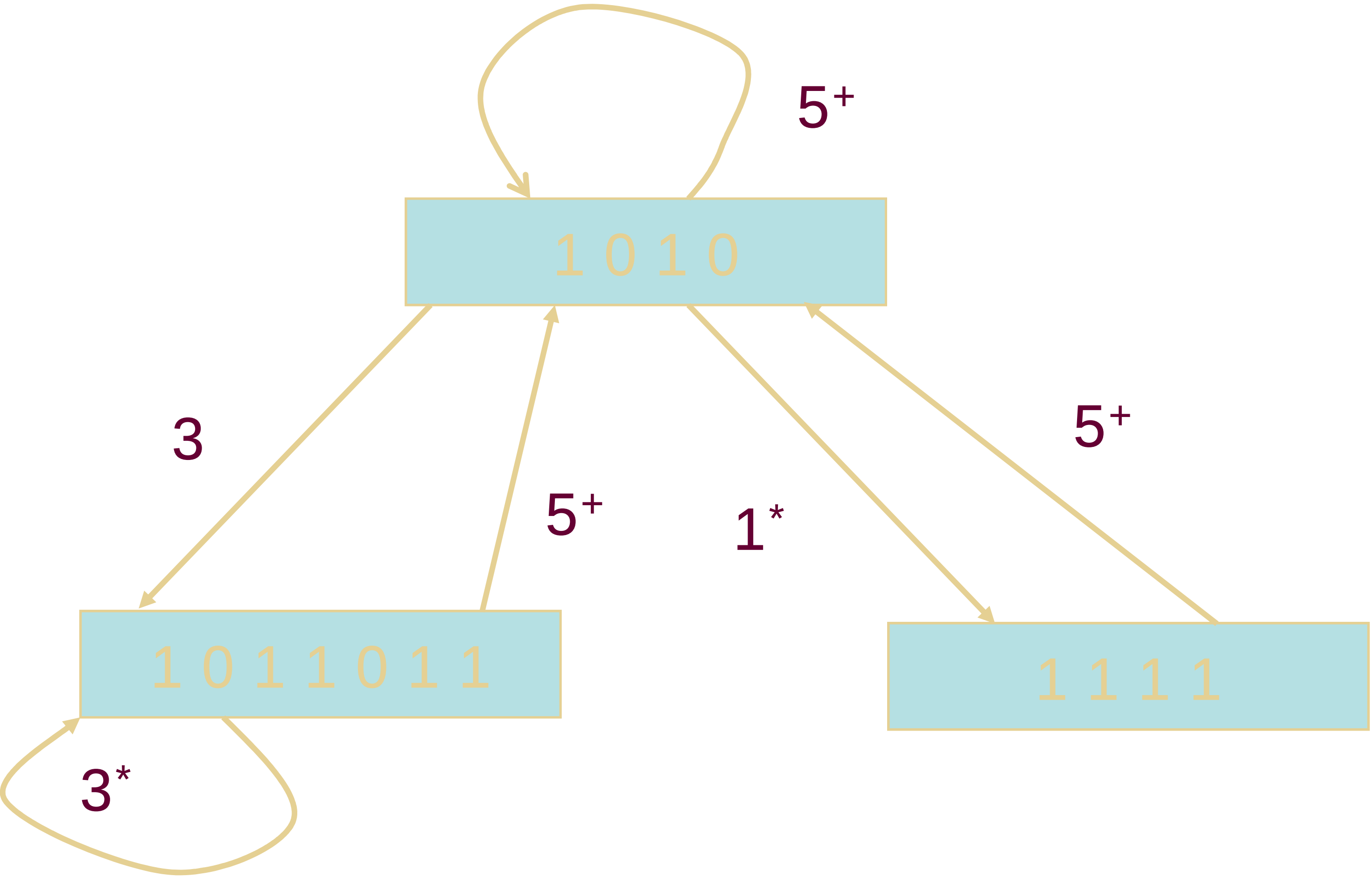
- Greedy Cycles → simple cycles whose edges are all made with minimum latencies from their respective starting states

  (1,8), (3) → one of them is MAL

MAL

- Minimum Average latency
- At least one of the greedy cycles will lead to the MAL
- Consider state diagram for Y, MAL is 3 (See diagram)

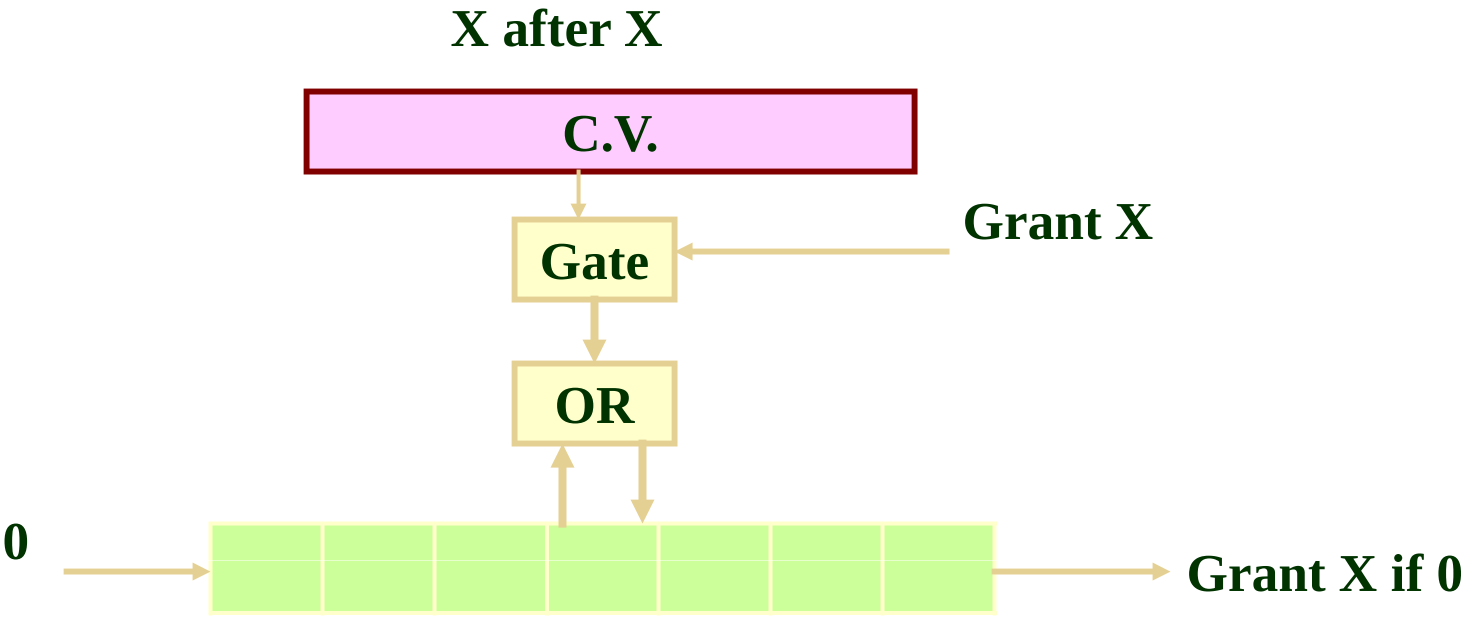
State Diagram for Y



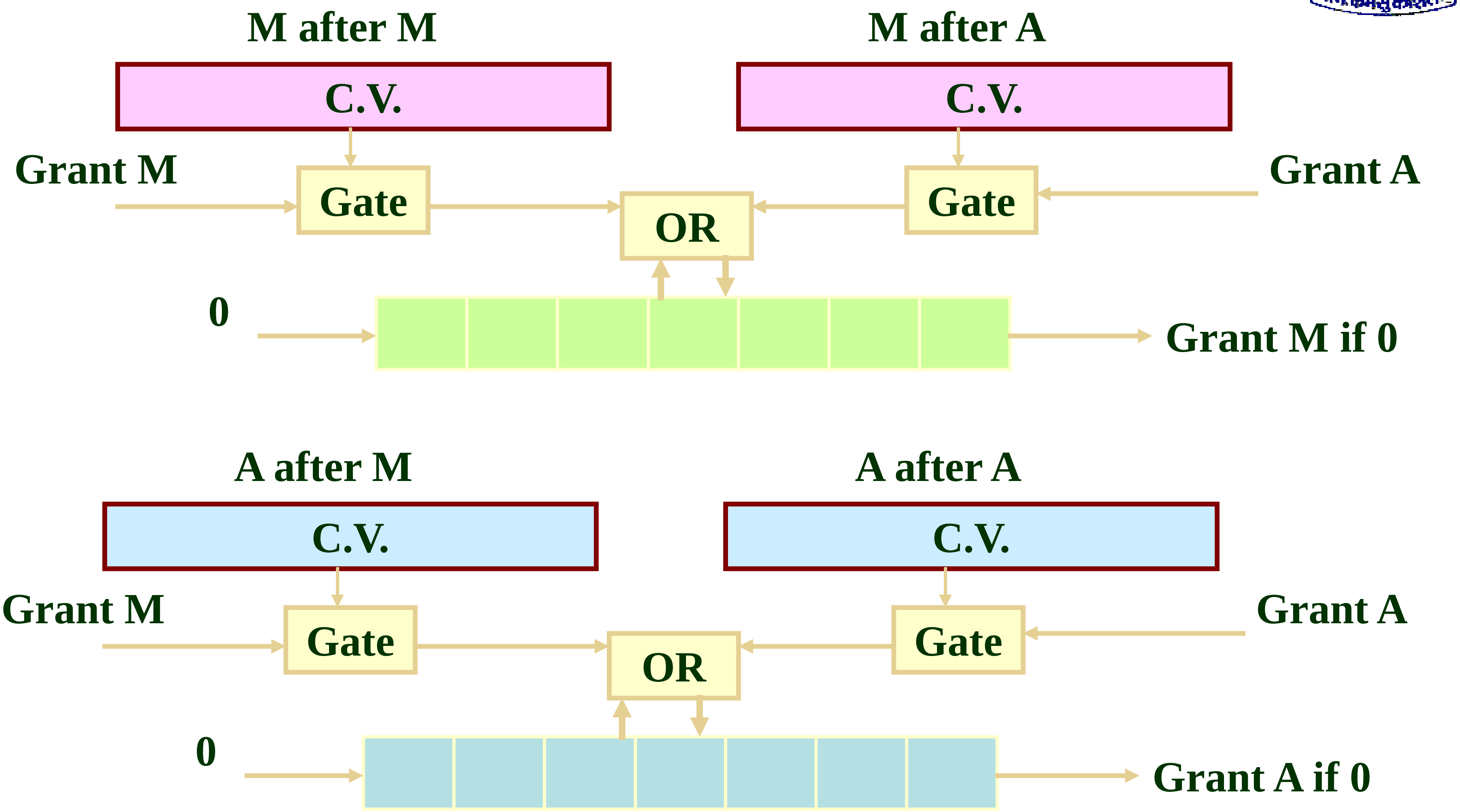
Bounds on the MAL

- MAL is lower bounded by the maximum number of checkmarks in any row of the reservation table. (Shar, 1972)
- MAL is lower than or equal to the average latency of any greedy cycle in the state diagram. (Shar, 1972)
- The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bund on the MAL. (Shar, 1972)

Single Function Controller



Controller for a dual-function pipeline

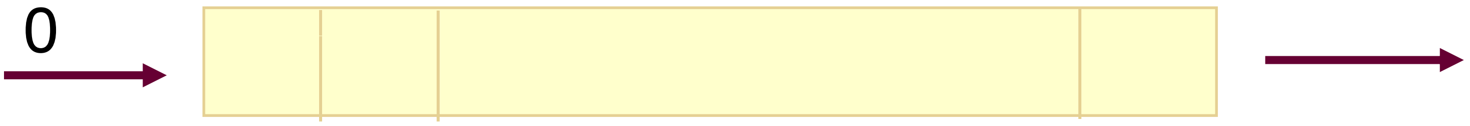


State Diagram

- It specifies the permissible state transitions among successive initiations
- Collision vector corresponds to the initial state at time t = 1 (initial collision vector)
- The next state comes at time t + p, where p is a permissible latency in the range 1 <= p < m

Right Shift Register

The next state can be obtained with the help of an m-bit shift register



0

1 Collision

0

Safe to allow an initiation

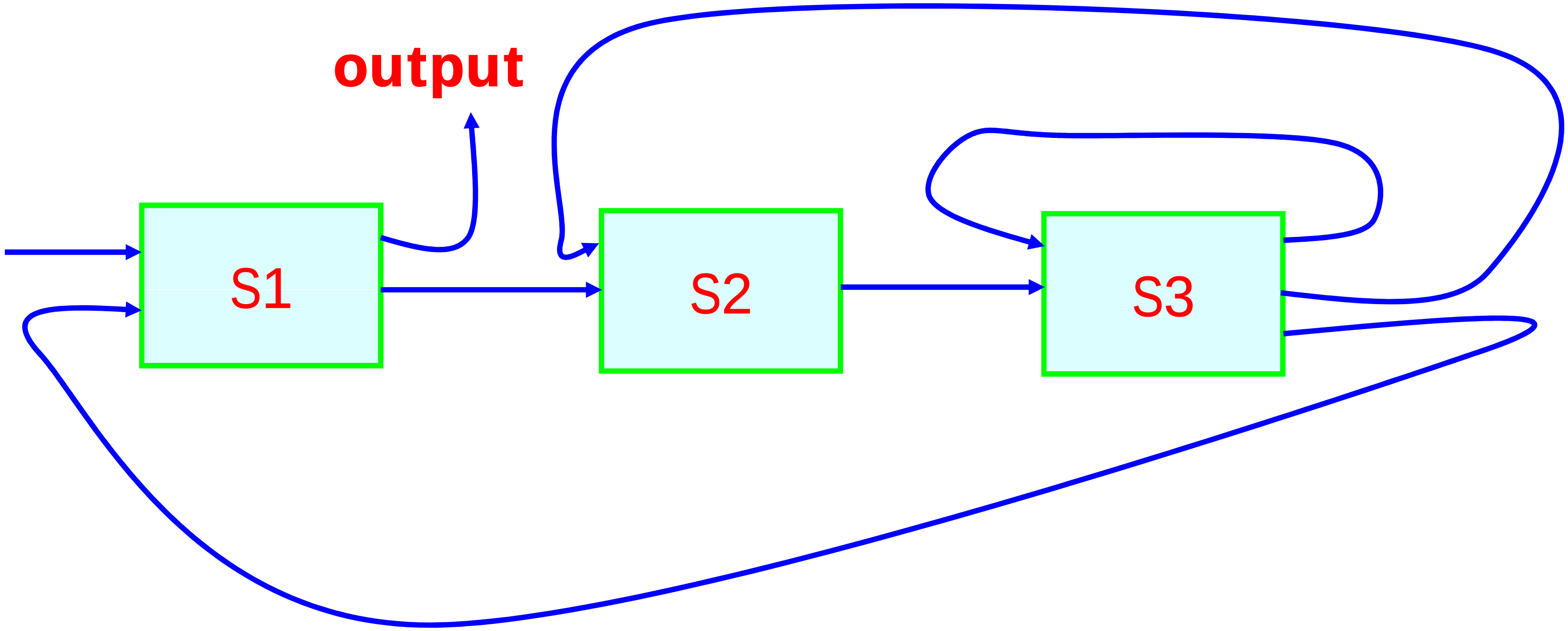
Each 1-bit shift corresponds to increase in the latency by 1

Delay Insertion

- The purpose is to modify the reservation table, yielding a new collision vector

- This may lead to a modified state diagram, which may produce greedy cycles meeting the lower bound on MAL

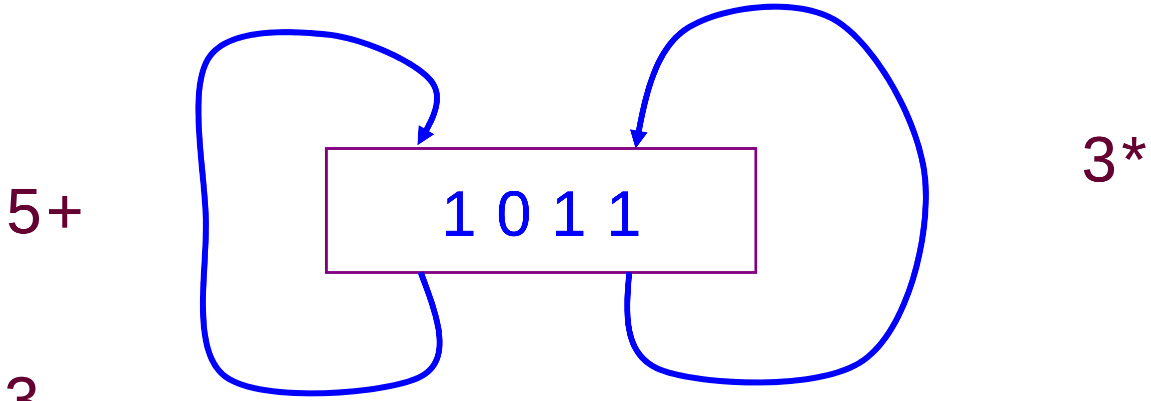
Example



Reservation Tables

	1	2	3	4	5
S1	X				X
S2		X		X	
S3			X	X	

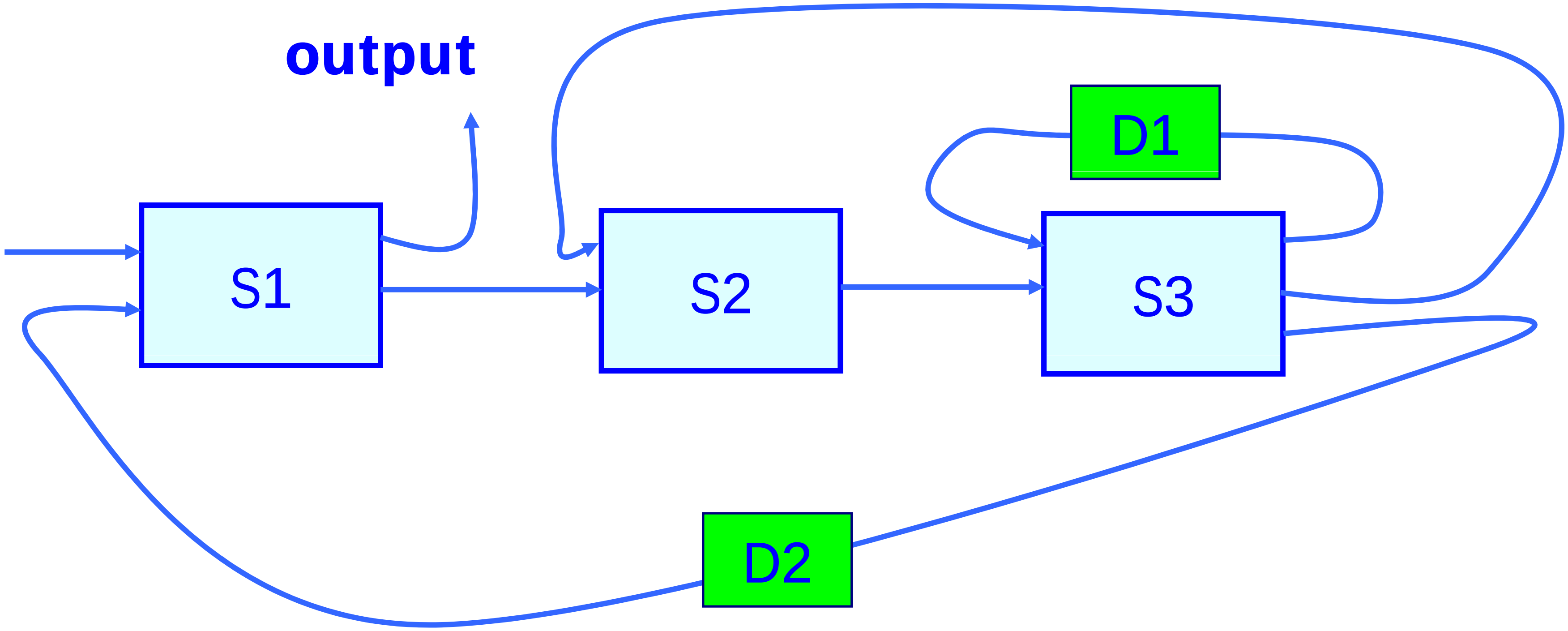
Forbidden Latencies: 1, 2, 4
C.V. → 1 0 1 1



MAL = 3

State Diagram

Example (Cont.)



Example (Cont.)

	1	2	3	4	5	6	7
S1	X						X
S2		X		X			
S3			X		X		
D1				X			
D2						X	

Forbidden: 2, 6

C.V. → 1 0 0 0 1 0

Hazards

Hazards

Hazards: problems due to pipelining

Hazard types:

- Structural
  - same resource is needed multiple times in the same cycle
- Data
  - data dependencies limit pipelining
- Control
  - next executed instruction may not be the next specified instruction

Structural hazards

Examples:

- Two accesses to a single ported memory
- Two operations need the same function unit at the same time
- Two operations need the same function unit in successive cycles, but the unit is not pipelined

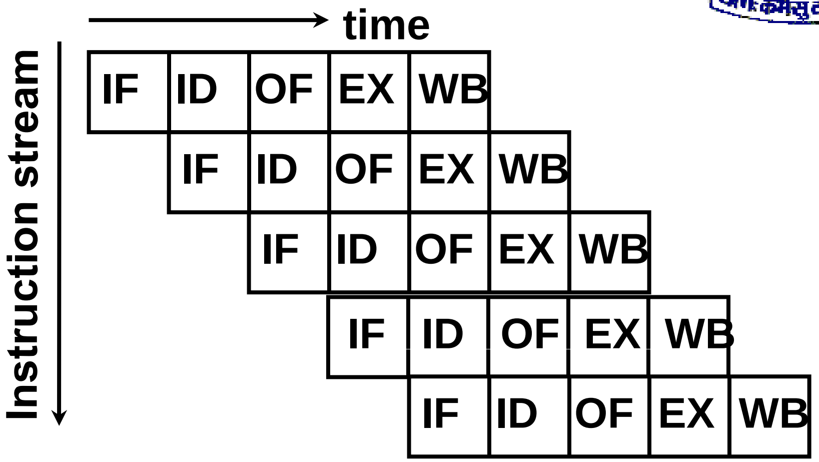
Solutions:

- stalling
- add more hardware

Structural hazards

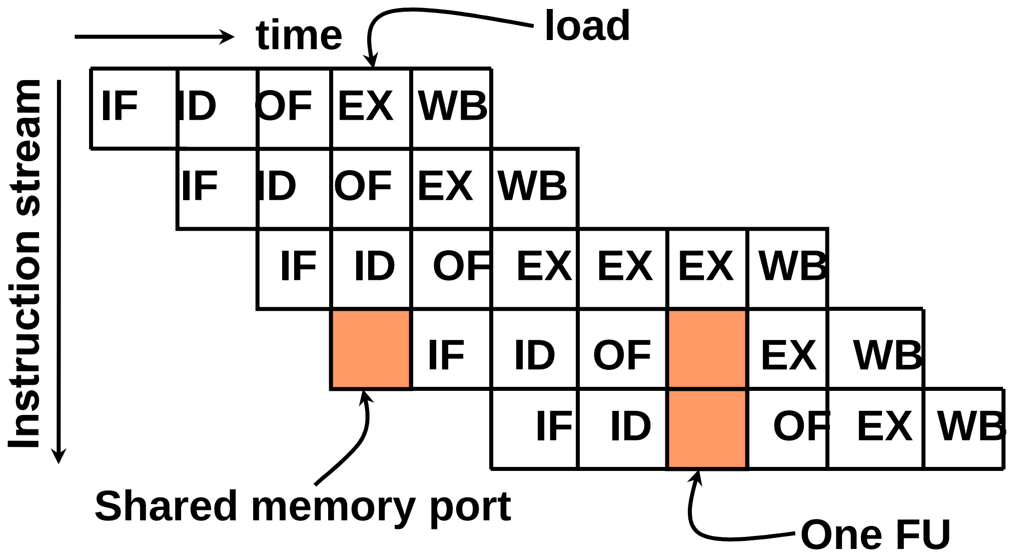
Simple pipelining diagram (but not MIPS!):

- IF: instruction fetch
- ID: instruction decode
- OF: operand fetch
- EX: execute stage(s)
- WB: write back



IF	ID	OF	EX	WB				
	IF	ID	OF	EX	WB			
		IF	ID	OF	EX	WB		
			IF	ID	OF	EX	WB	
				IF	ID	OF	EX	WB

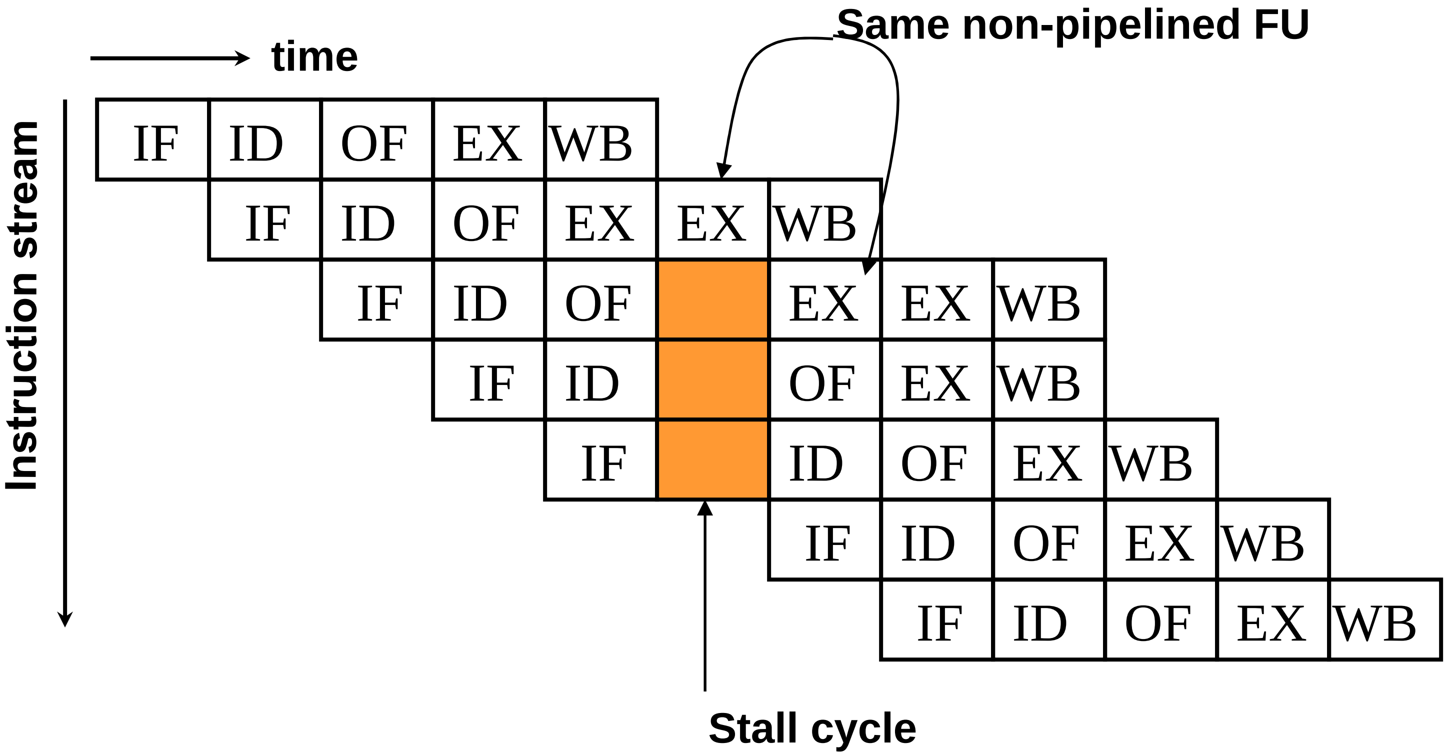
Pipeline stalls due to lack of resources:



IF	ID	OF	EX	WB						
	IF	ID	OF	EX	WB					
		IF	ID	OF	EX	EX	EX	WB		
			▨	IF	ID	OF	▨	EX	WB	
					IF	ID	▨	OF	EX	WB

Structural hazards

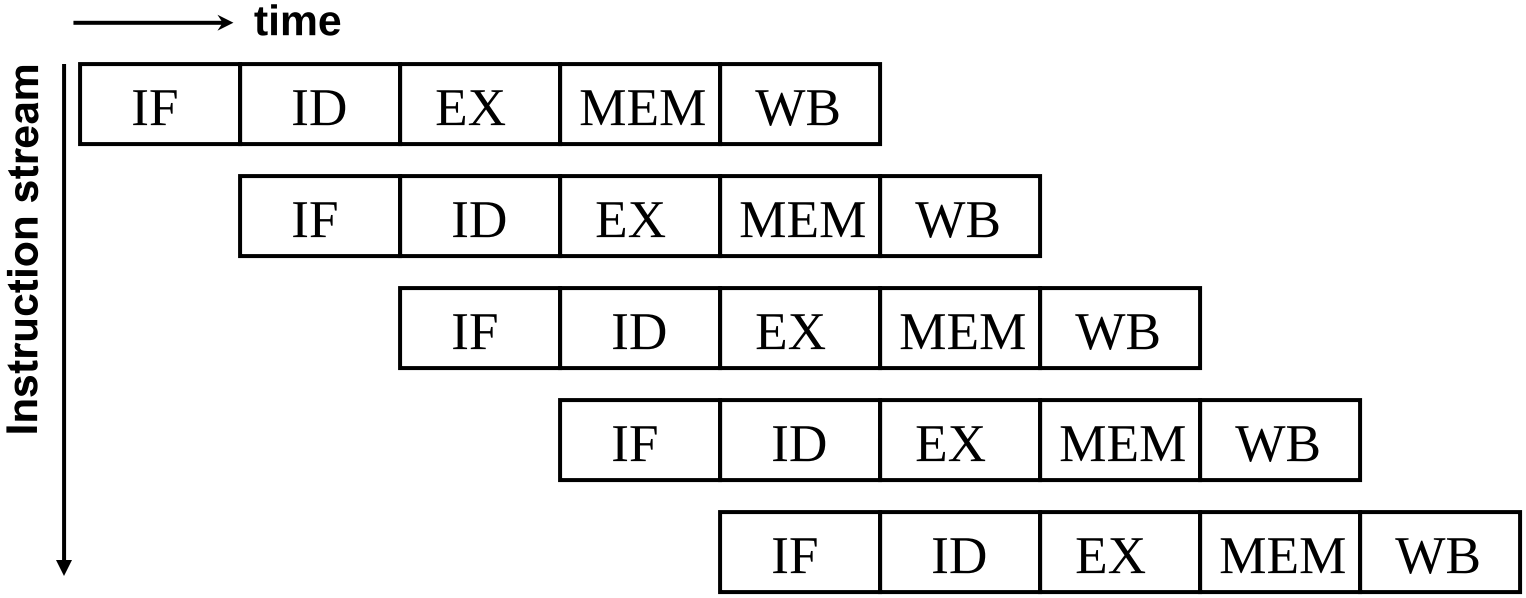
Non-pipelined units



IF	ID	OF	EX	WB							
	IF	ID	OF	EX	EX	WB					
		IF	ID	OF	▨	EX	EX	WB			
			IF	ID	▨	OF	EX	WB			
				IF	▨	ID	OF	EX	WB		
						IF	ID	OF	EX	WB	
							IF	ID	OF	EX	WB

Structural hazards on MIPS

Q: Do we have structural hazards on our simple MIPS pipeline?



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Data hazards

- Data dependencies:
  - RaW (read-after-write)
  - WaW (write-after-write)
  - WaR (write-after-read)
- Hardware solution:
  - Forwarding / Bypassing
  - Detection logic
  - Stalling
- Software solution: Scheduling

Data dependences

Three types: RaW, WaR and WaW

```
add r1, r2, 5       ; r1 := r2+5
sub r4, r1, r3      ; RaW of r1

add r1, r2, 5
sub r2, r4, 1       ; WaR of r2

add r1, r2, 5
sub r1, r1, 1       ; WaW of r1

st  r1, 5(r2)       ; M[r2+5] := r1
ld  r5, 0(r4)       ; RaW if 5+r2 = 0+r4
```

WaW and WaR do not occur in simple pipelines, but they limit scheduling freedom!

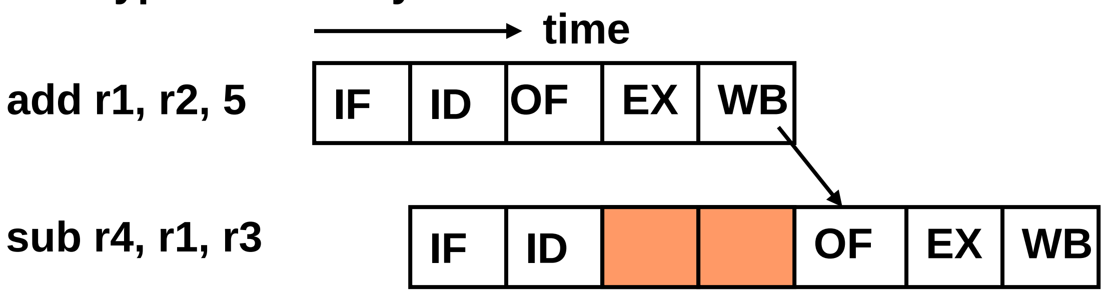
Problems for your compiler and Pentium!

⇒ use register renaming to solve this!

RaW dependence

```
add r1, r2, 5        ;r1:= r2+5
sub r4, r1, r3       ;RaW of r1
```

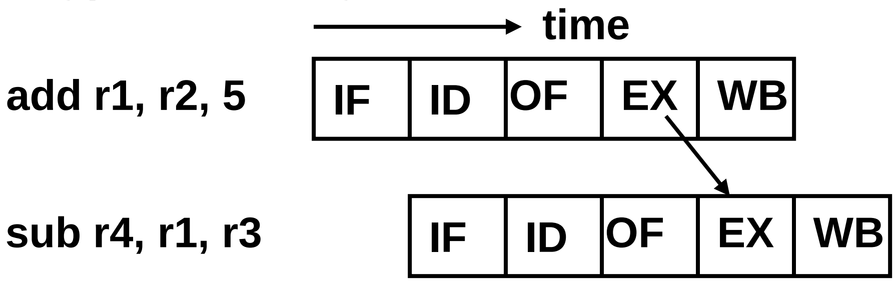
Without bypass circuitry



time

add r1, r2, 5	IF	ID	OF	EX	WB			
sub r4, r1, r3		IF	ID	🟧	🟧	OF	EX	WB

With bypass circuitry

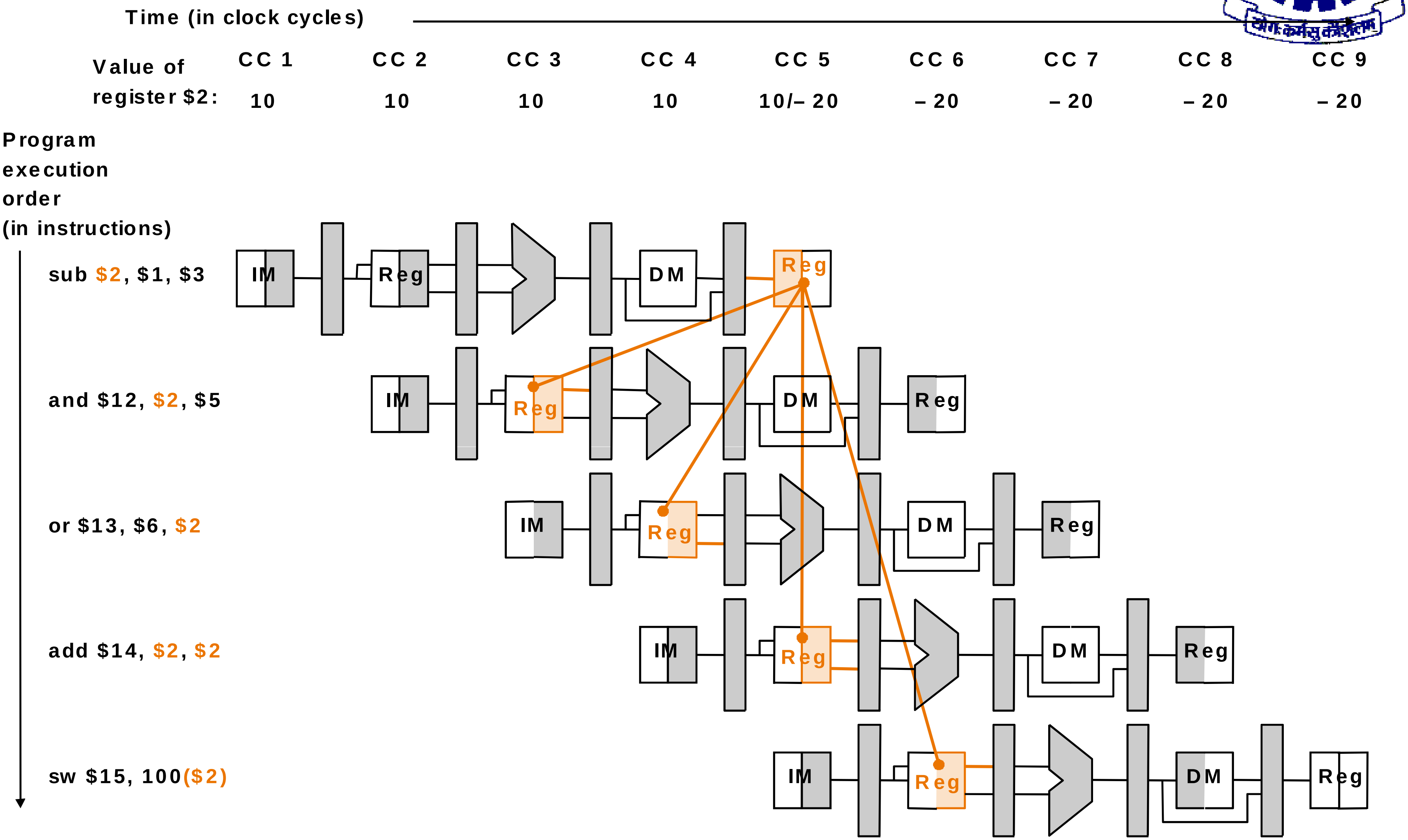


time

add r1, r2, 5	IF	ID	OF	EX	WB	
sub r4, r1, r3		IF	ID	OF	EX	WB

Saves two cycles

RaW on MIPS pipeline



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

Program execution order (in instructions)

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

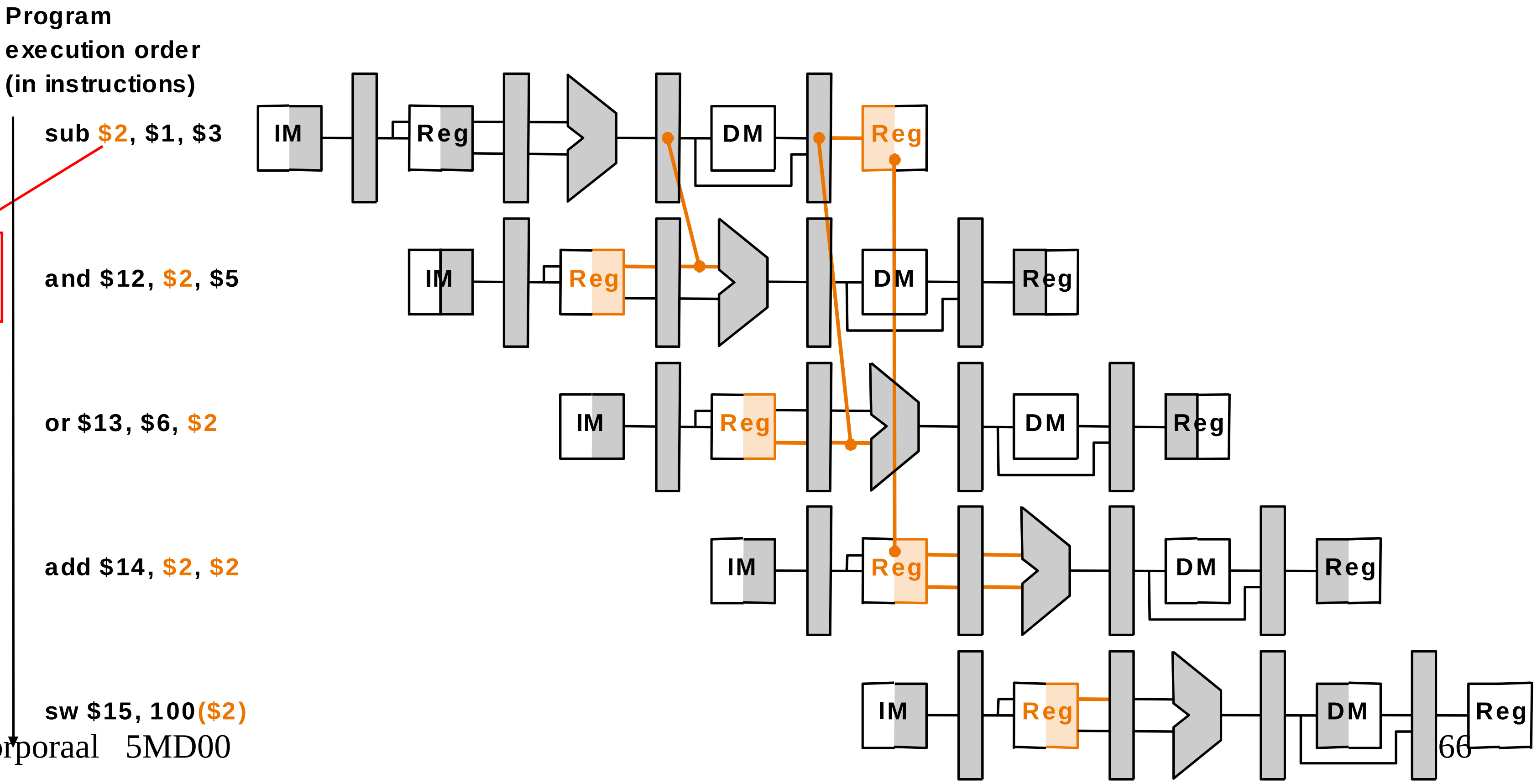
Forwarding

Use temporary results, don't wait for them to be written

- register file forwarding to handle read/write to same register
- ALU 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



Program execution order (in instructions)

sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

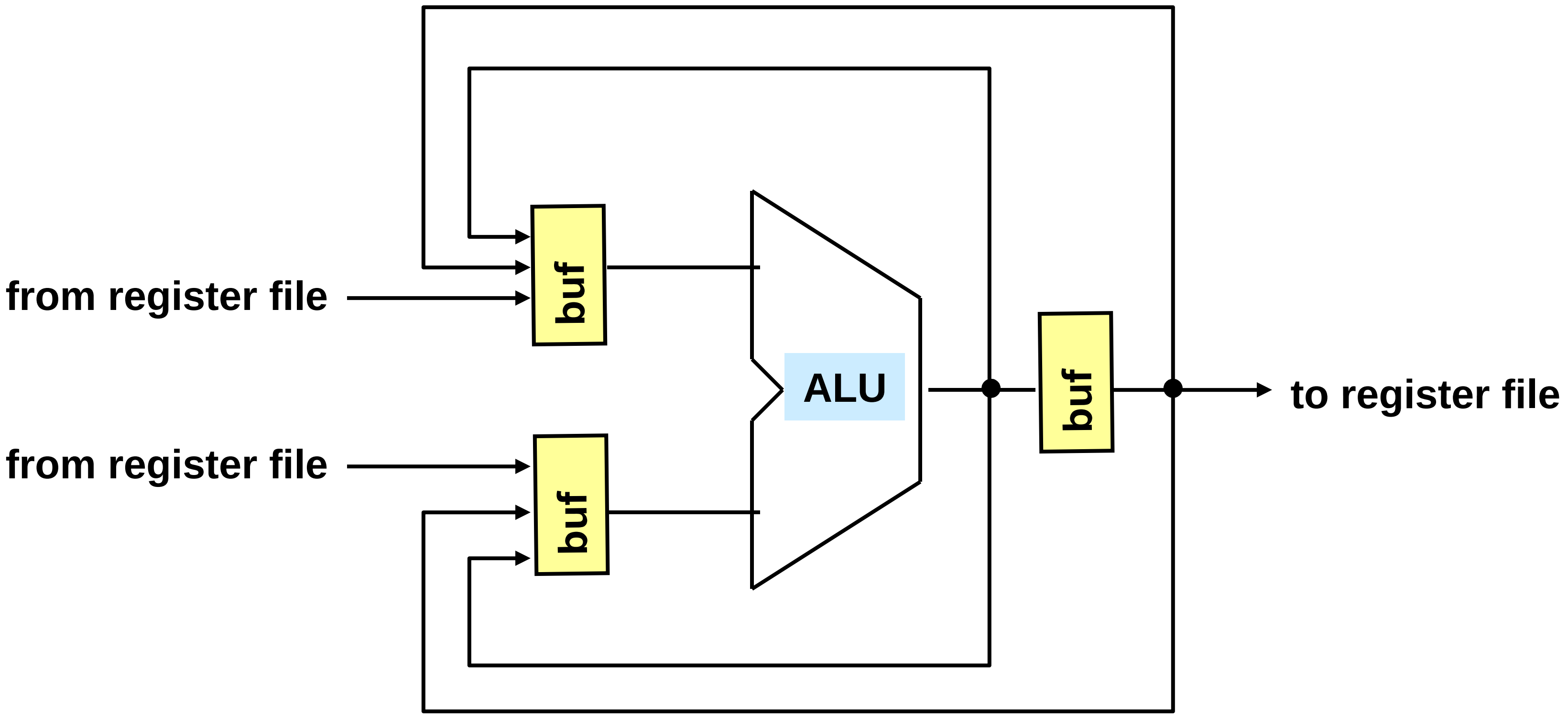
add $14, $2, $2

sw $15, 100($2)

What if this $2 was $13?

Forwarding hardware

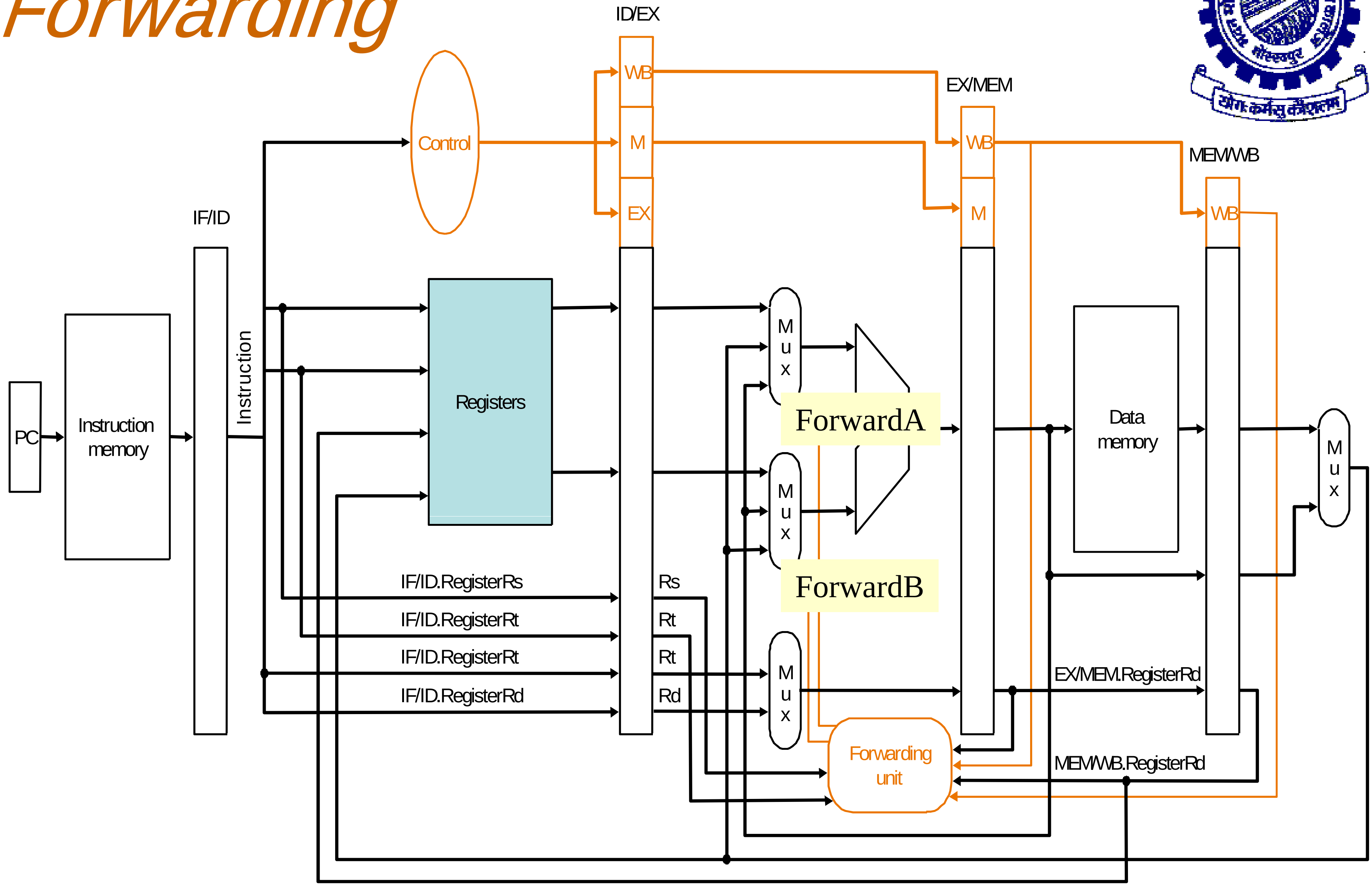
ALU forwarding circuitry principle:



Note: there are two options

- buf - ALU – bypass – mux - buf
- buf - bypass – mux – ALU - buf

Forwarding


ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
Control
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
M u x
M u x
M u x
ForwardA
ForwardB
Data memory
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit

Forwarding check

- Check for matching register-ids:
- For each source-id of operation in the EX-stage check if there is a matching pending dest-id

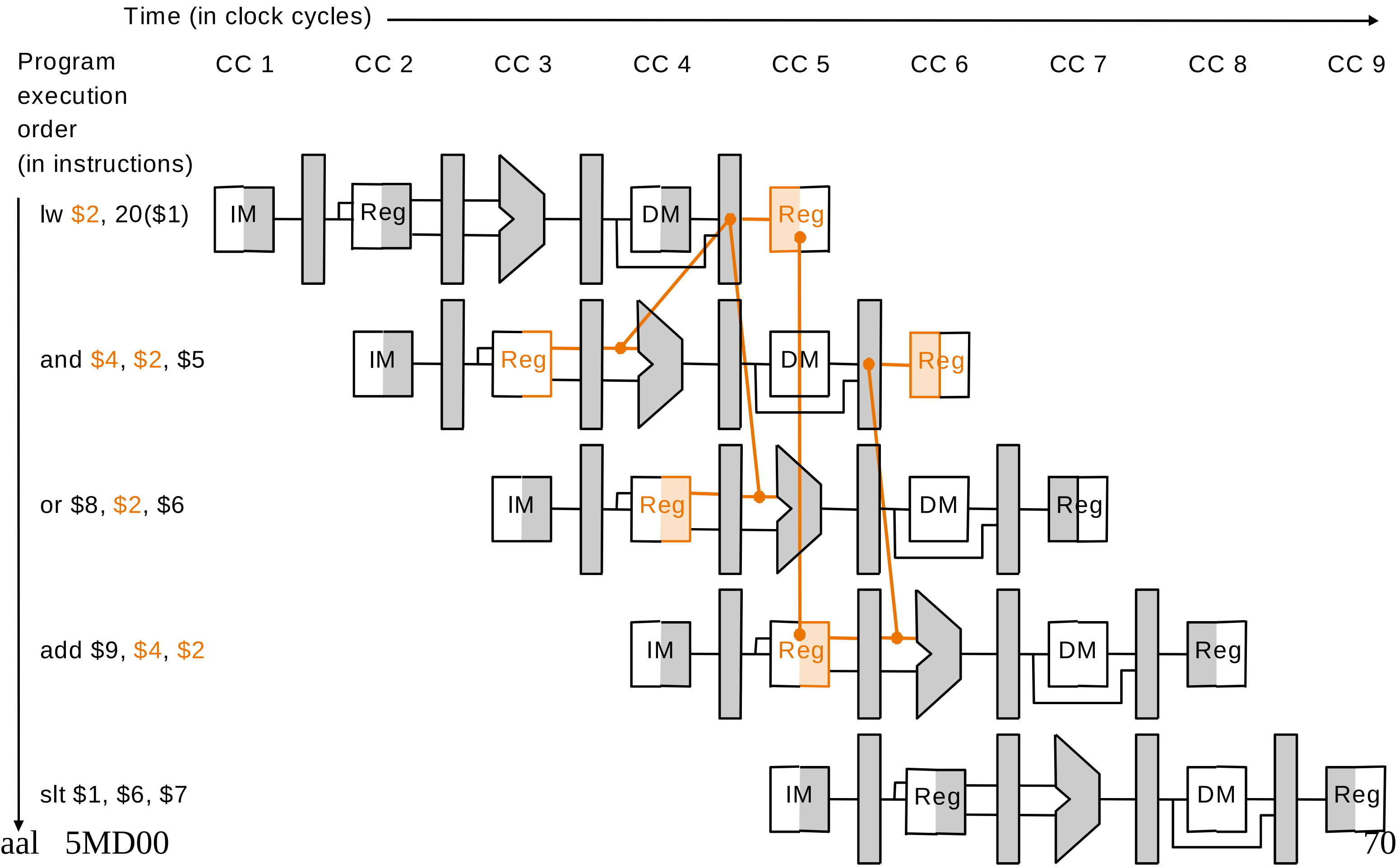
Example:

```
if (EX/MEM.RegWrite) ∧
   (EX/MEM.RegisterRd ≠ 0) ∧
   (EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 10
```

Q. How many comparators do we need?

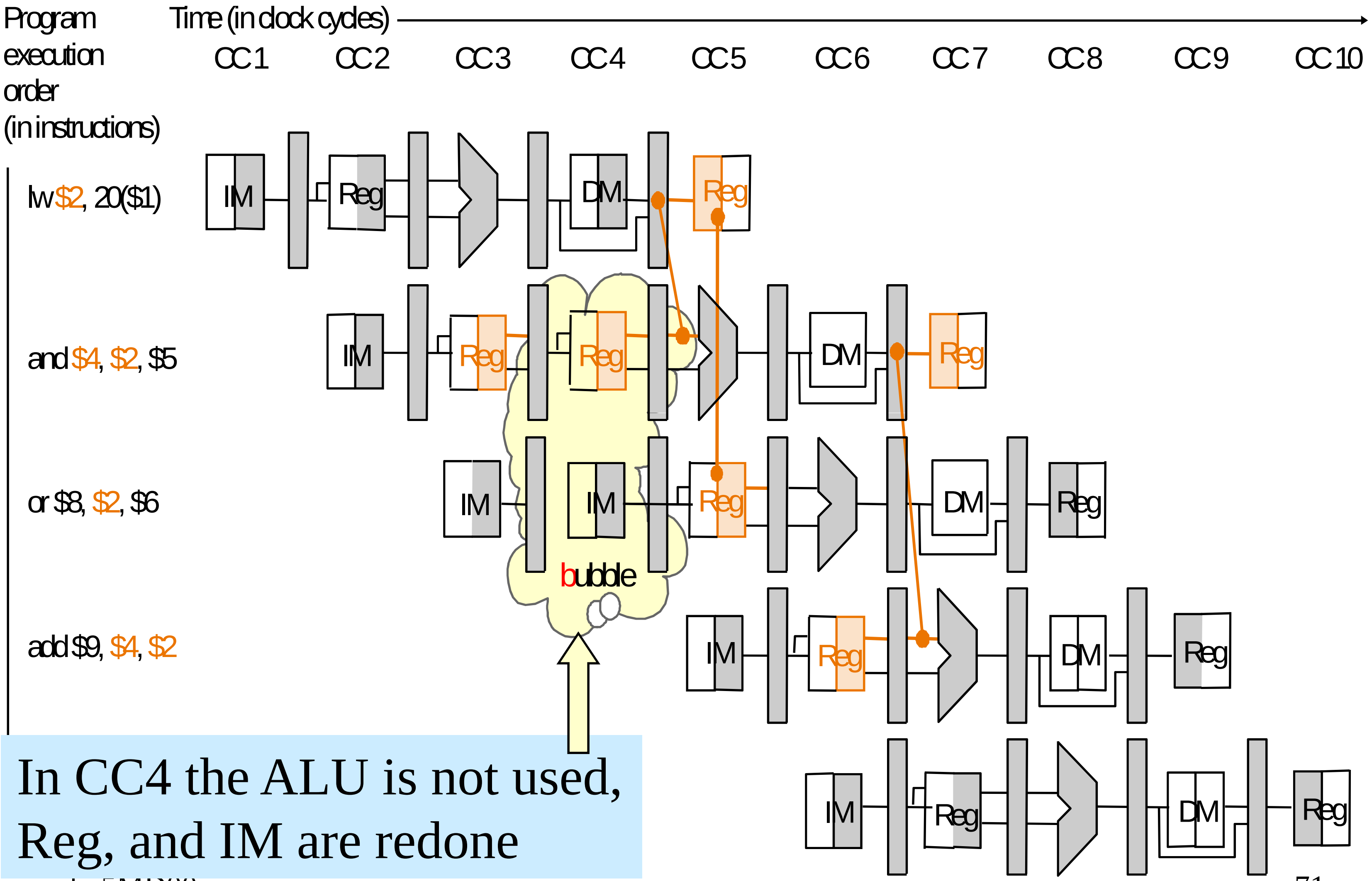
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read register r following a load to the same r
- Need a hazard detection unit to “stall” the load instruction



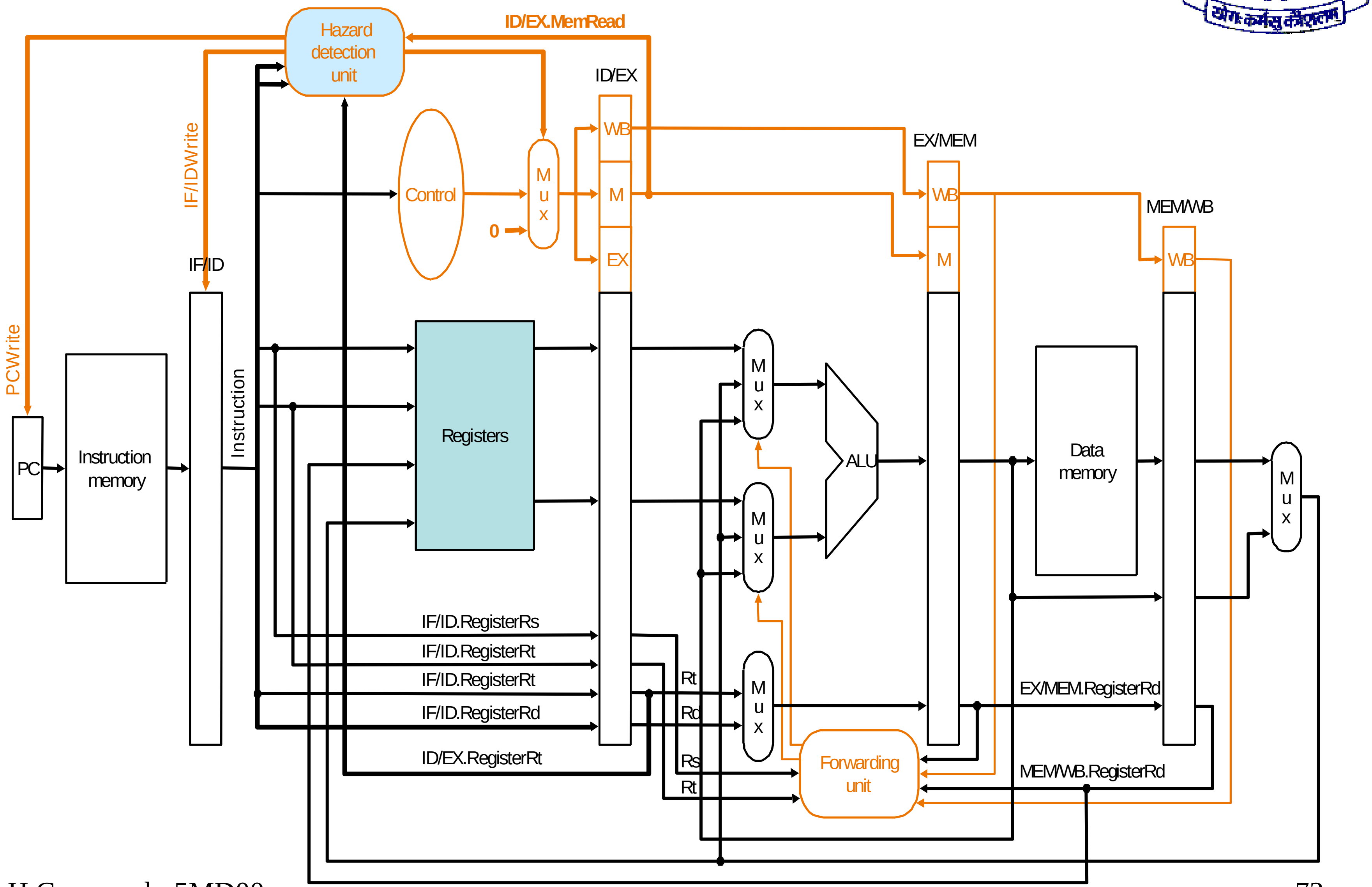
Stalling

We can stall the pipeline by keeping an instruction in the same stage



In CC4 the ALU is not used, Reg, and IM are redone

Hazard Detection Unit



Software only solution?

- Have compiler guarantee that no hazards occur
- Example: where do we insert the “NOPs” ?

```
sub   $2,   $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $13, 100($2)
```

```
sub   $2,   $1, $3
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
nop
sw    $13, 100($2)
```

- Problem: this really slows us down!

Control hazards

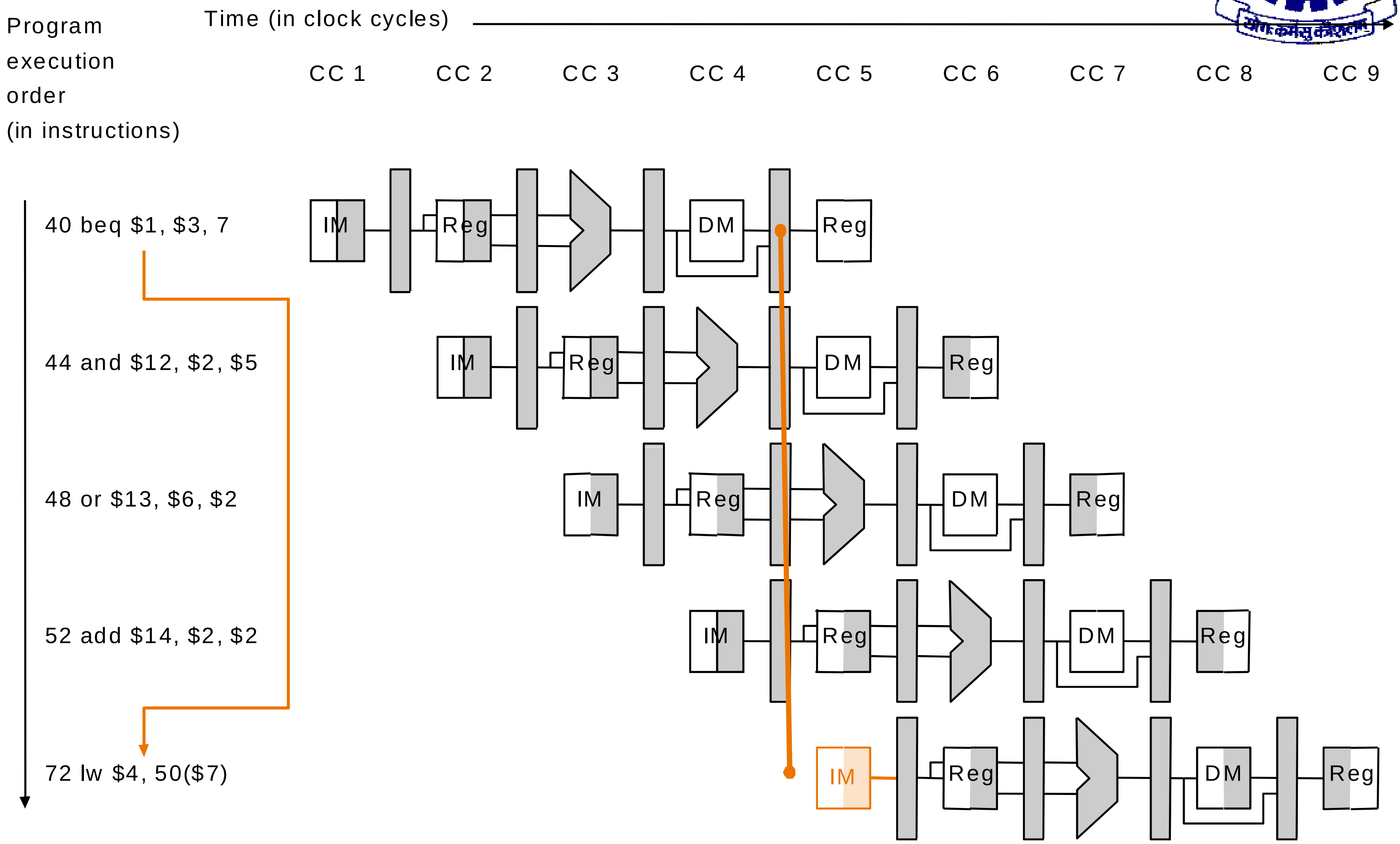
- Control operations may change the sequential flow of instructions
  - branch
  - jump
  - call (jump and link)
  - return
  - (exception/interrupt and rti / return from interrupt)

Control hazard: Branch

Branch actions:

- Compute new address
- Determine condition
- Perform the actual branch (if taken): PC := new address

Branch example



Program execution order (in instructions)

Time (in clock cycles)

CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9

40 beq $1, $3, 7

44 and $12, $2, $5

48 or $13, $6, $2

52 add $14, $2, $2

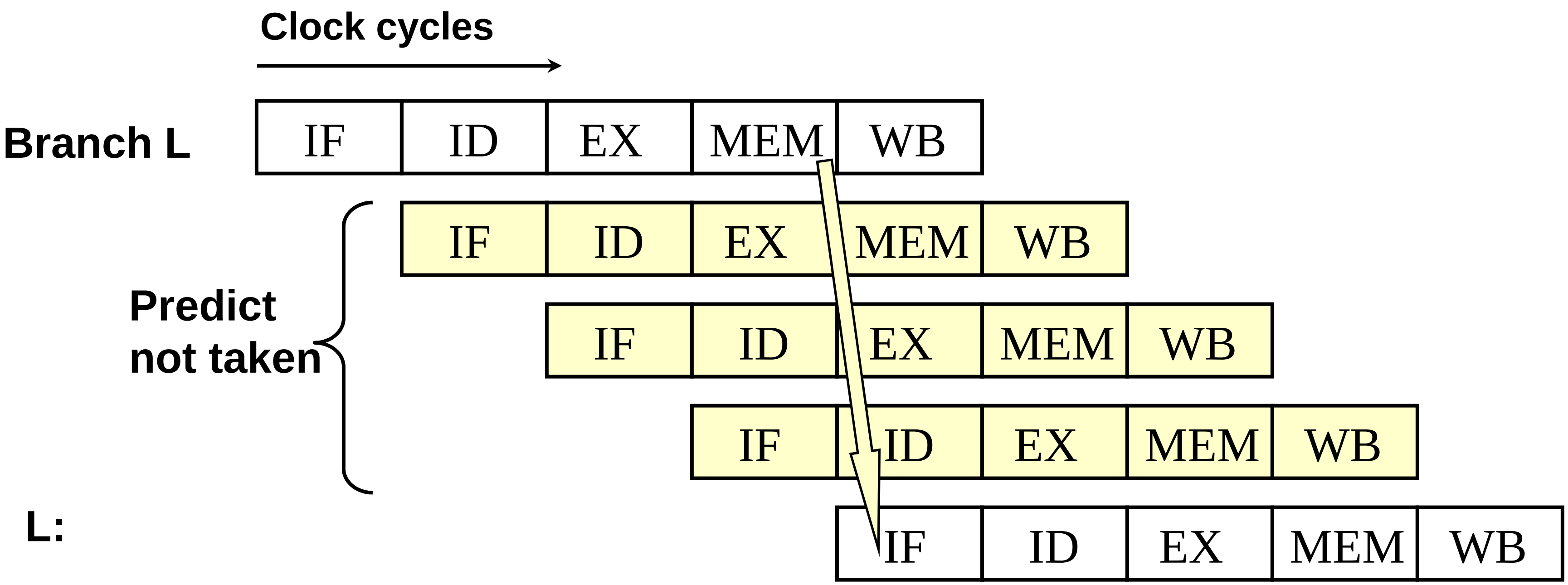
72 lw $4, 50($7)

Branching

Squash pipeline:

- When we decide to branch, other instructions are in the pipeline!
- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong

Branch with predict not taken



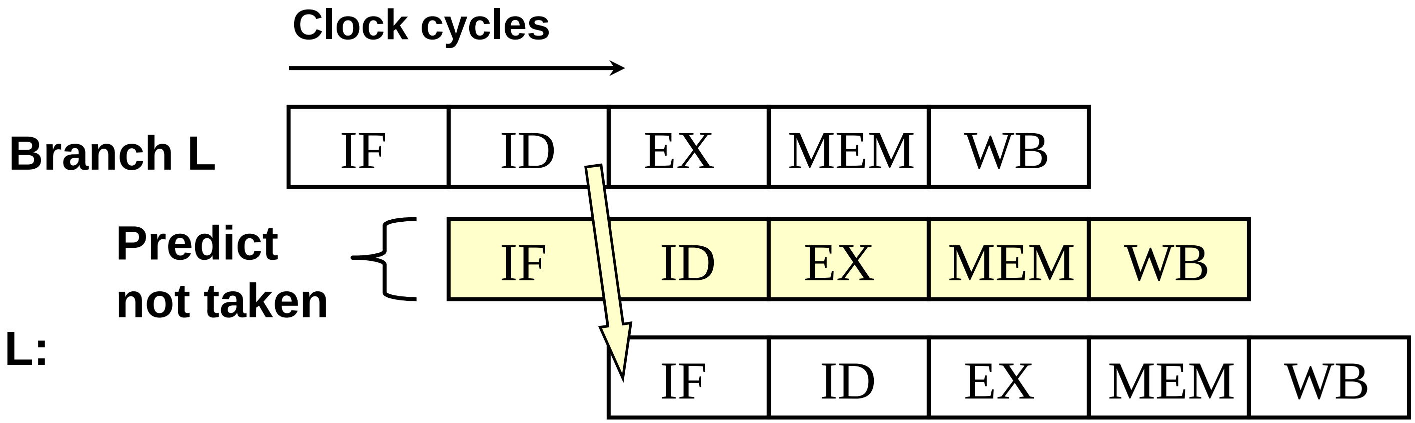
Clock cycles

Branch L	IF	ID	EX	MEM	WB				
Predict not taken		IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	
L:					IF	ID	EX	MEM	WB

Branch speedup

- Earlier address computation
- Earlier condition calculation

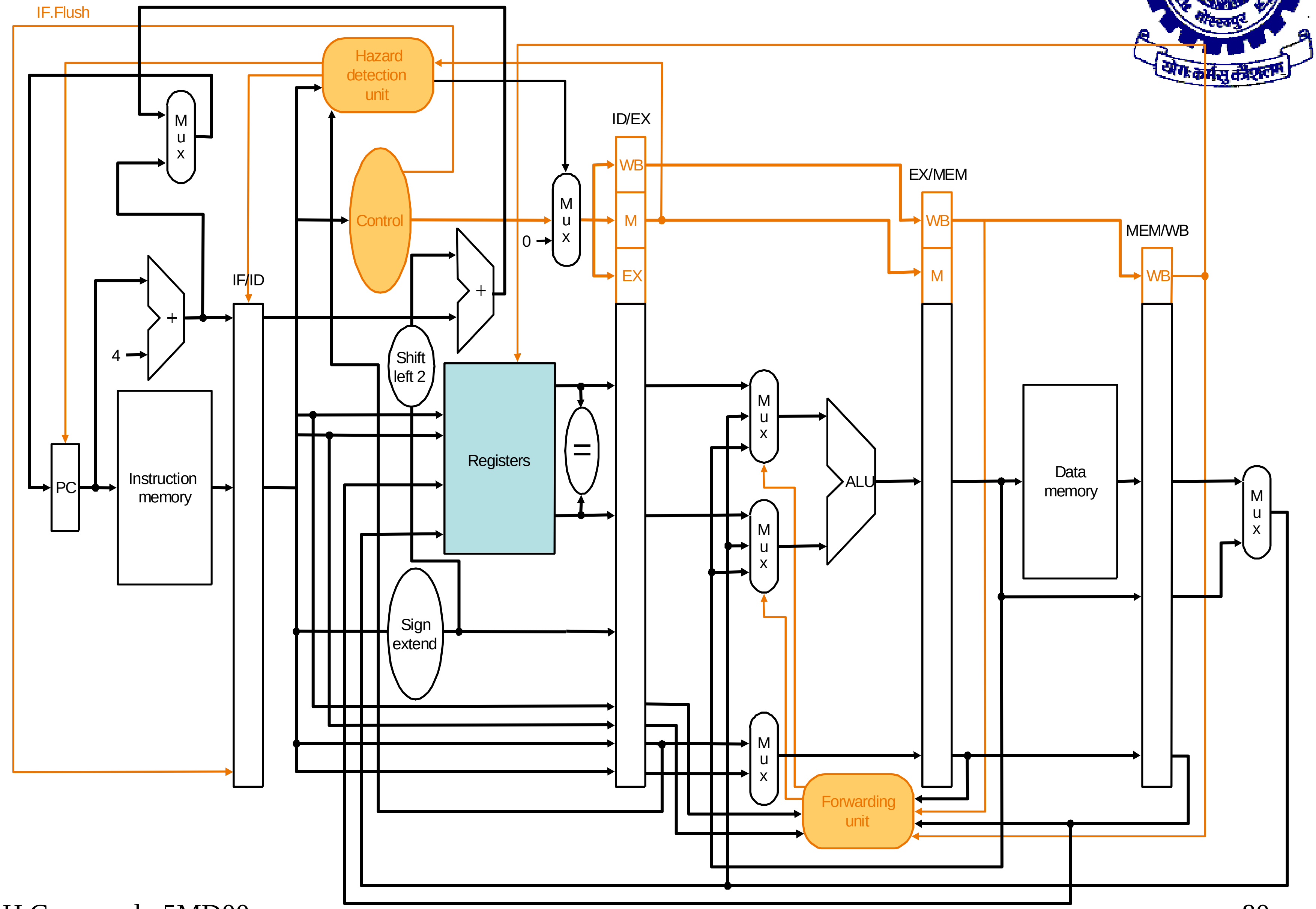
- Put both in the ID pipeline stage
  - adder
  - comparator



Clock cycles →

Branch L	IF	ID	EX	MEM	WB		
Predict not taken		IF	ID	EX	MEM	WB	
L:			IF	ID	EX	MEM	WB

Improved branching / flushing IF/ID



Exception support

Types of exceptions:

- Overflow
- I/O device request
- Operating system call
- Undefined instruction
- Hardware malfunction
- Page fault

- Precise exception:
  - finish previous instructions (which are still in the pipeline)
  - flush excepting and following instructions, redo them after handling the exception(s)

Exceptions

Changes needed for handling overflow exception of an operation in EX stage (see book for details) :

- Extend PC input mux with extra entry with fixed address
- Add EPC register recording the ID/EX stage PC
  - this is the address of the next instruction !
- Cause register recording exception type

E.g., in case of overflow exception insert 3 bubbles; flush the following stages:

- IF/ID stage
- ID/EX stage
- EX/MEM stage

Scheduling, why?

Let’s look at the execution time:

$$T_{execution} = N_{cycles} \times T_{cycle}$$

$$= N_{instructions} \times CPI \times T_{cycle}$$

Scheduling may reduce $T_{execution}$

- Reduce *CPI* (cycles per instruction)
  - early scheduling of long latency operations
  - avoid pipeline stalls due to structural, data and control hazards
  - allow $N_{issue} > 1$ and therefore $CPI < 1$
- Reduce $N_{instructions}$
  - compact many operations into each instruction (VLIW)

Scheduling data hazards: example 1

Try and avoid RaW stalls (in this case load interlocks)!
E.g., reorder these instructions:

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

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

Scheduling data hazards example 2

Avoiding RaW stalls:

Reordering instructions for following program

(by you or the compiler)

Code:

```
a = b + c
d = e - f
```

Unscheduled code:

```
Lw   R1,b
Lw   R2,c
Add  R3,R1,R2       interlock
Sw   a,R3
Lw   R1,e
Lw   R2,f
Sub  R4,R1,R2       interlock
Sw   d,R4
```

Scheduled code:

```
Lw   R1,b
Lw   R2,c
Lw   R5,e      extra reg. needed!
Add  R3,R1,R2
Lw   R2,f
Sw   a,R3
Sub  R4,R5,R2
Sw   d,R4
```

Scheduling control hazards

$$T_{execution} = N_{instructions} \text{ x } CPI \text{ x } T_{cycle}$$

$$CPI = CPI_{ideal} + f_{branch} \text{ x } P_{branch}$$

$$P_{branch} = N_{delayslots} \text{ x } miss\_rate$$

- Modern processors tend to have large branch penalty, $P_{branch,}$ due to:
  - many pipeline stages
  - multi-issue
- Note that penalties have larger effect when CPI_{ideal} is low

Scheduling control hazards

What can we do about control hazards and CPI penalty?

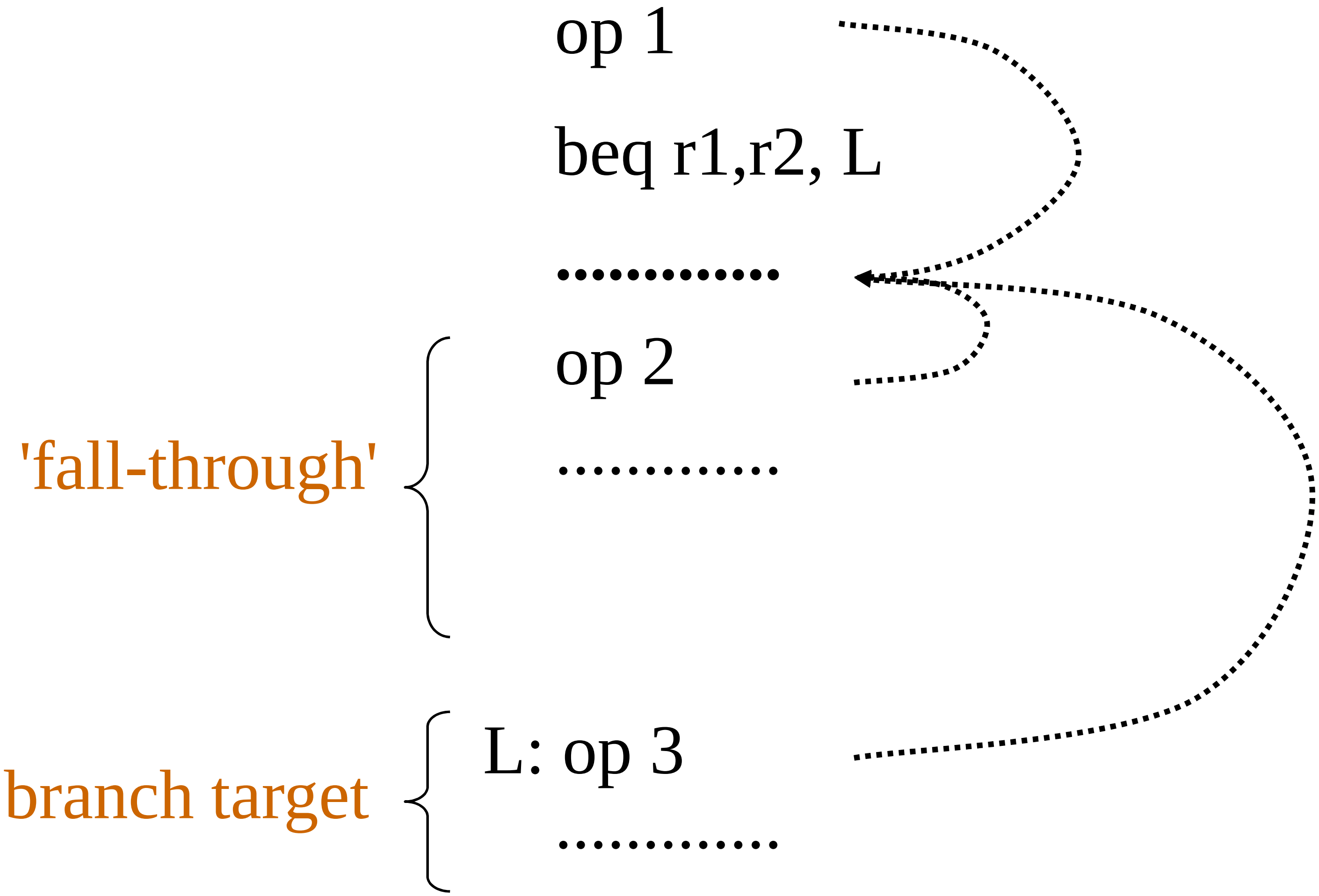
- Keep penalty P_{branch} low:
  - Early computation of new PC
  - Early determination of condition
  - Visible branch delay slots filled by compiler (MIPS)
- Branch prediction
- Reduce control dependencies (control height reduction) [Schlansker and Kathail, Micro'95]
- Remove branches: if-conversion
  - Conditional instructions: CMOVE, cond skip next
  - Guarding all instructions: TriMedia

Branch delay slot

- Add a branch delay slot:
  - the next instruction after a branch is always executed
  - rely on compiler to “fill” the slot with something useful
- Is this a good idea?
  - let's look how it works

Branch delay slot scheduling

Q. What to put in the delay slot?



op 1

beq r1,r2, L

....

'fall-through'

op 2

....

branch target

L: op 3

....

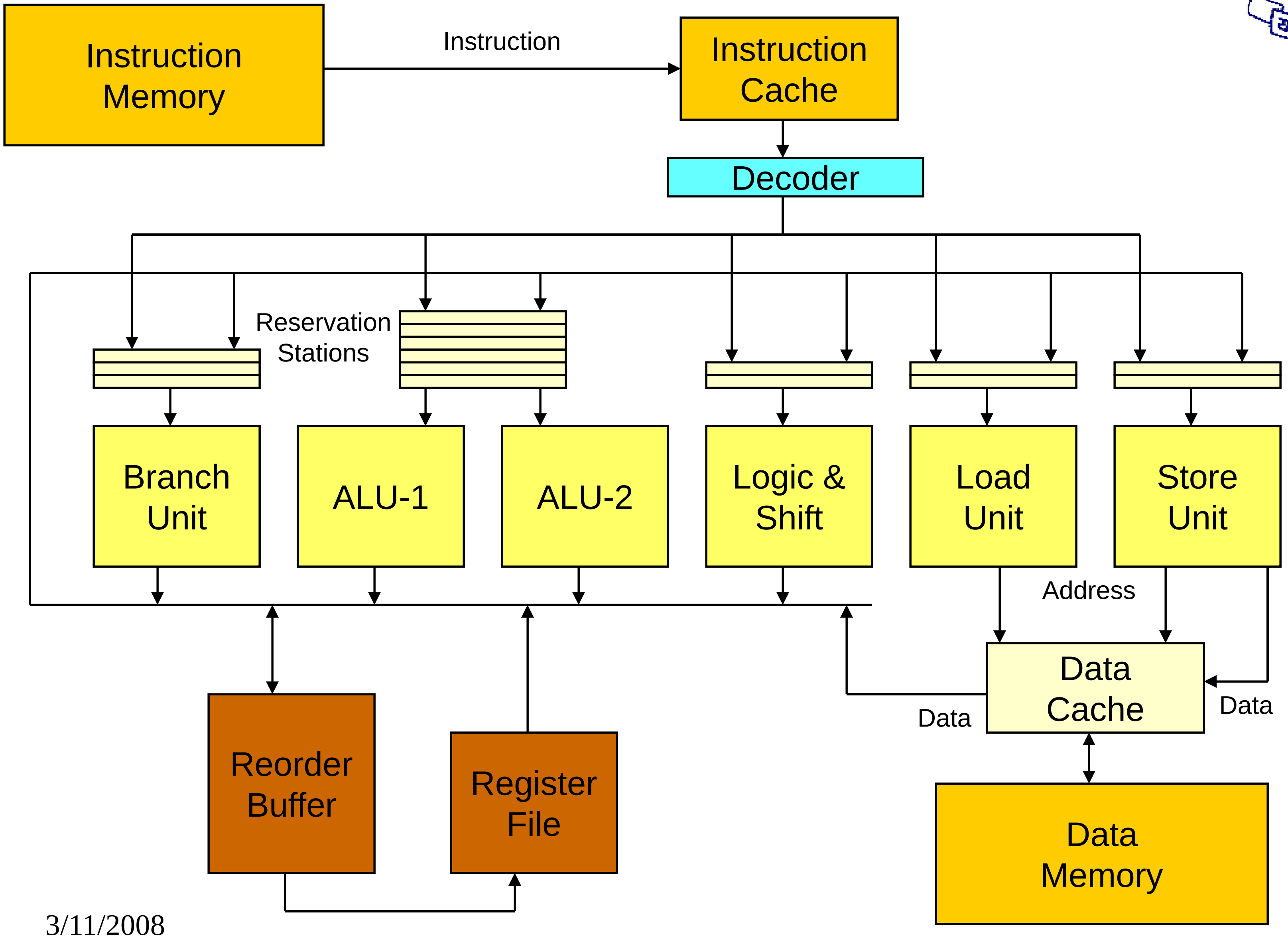
Summary

- Modern processors are (deeply) pipelined, to reduce T_{cycle} and aim at CPI = 1
- Hazards increase CPI
- Several software and hardware measure to avoid or reduce hazards are taken

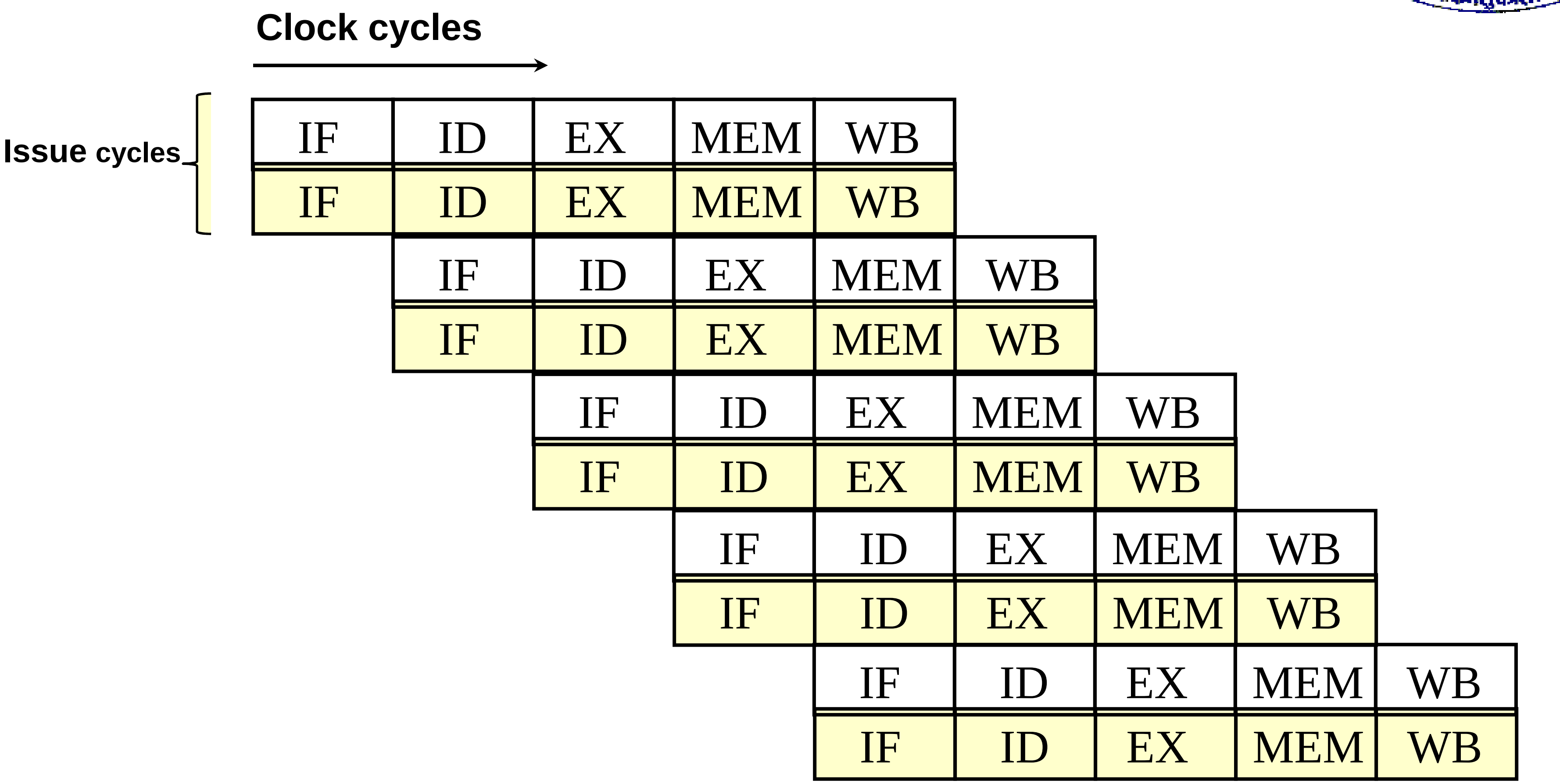
Not discussed, but important developments:

- Multi-issue further reduces CPI
- Branch prediction to avoid high branch penalties
- Dynamic scheduling
- In all cases: a scheduling compiler needed

Superscalar Concept

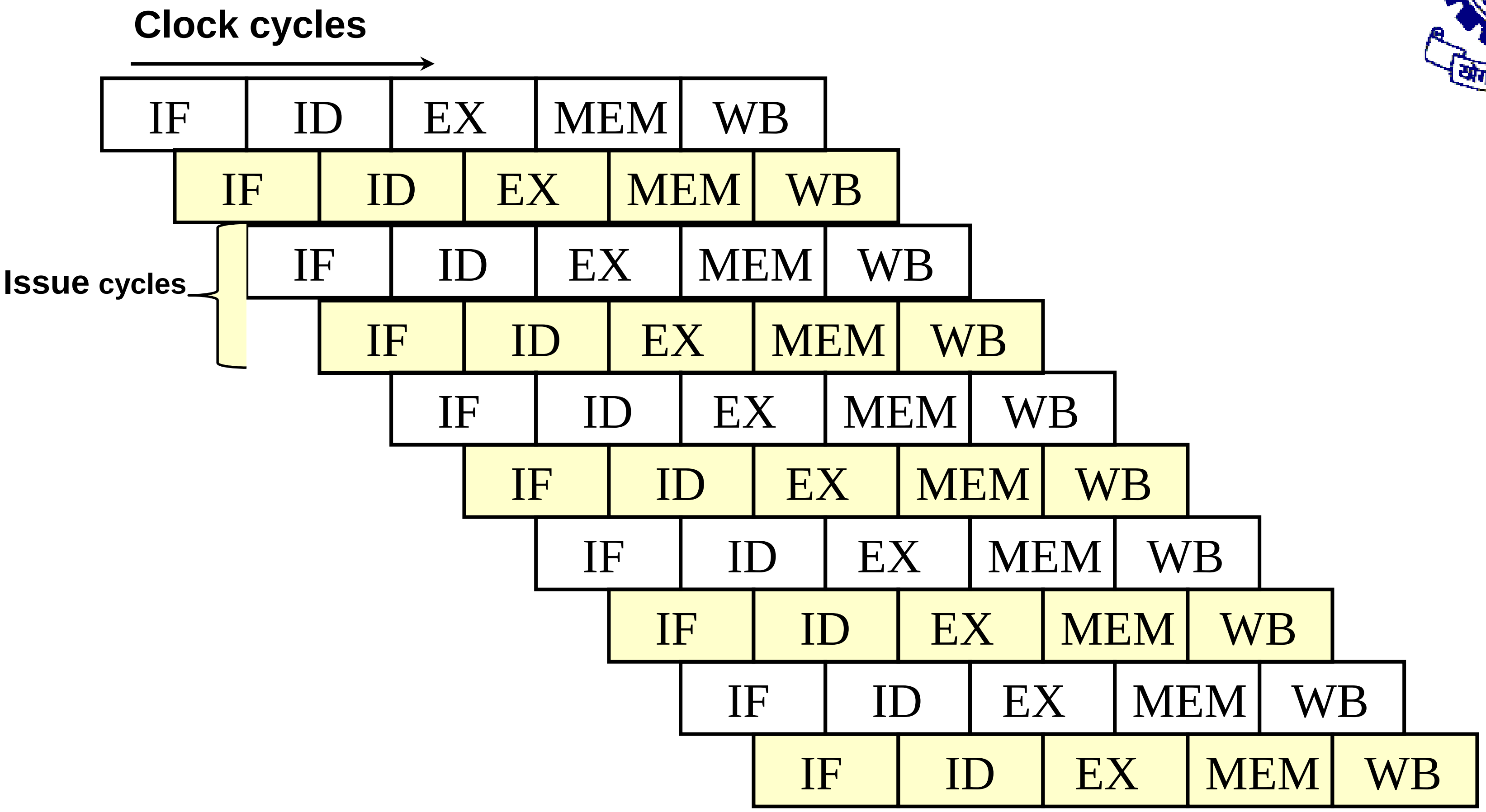


Superscalar Execution



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Superpipelined Execution



Clock cycles

Issue cycles

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

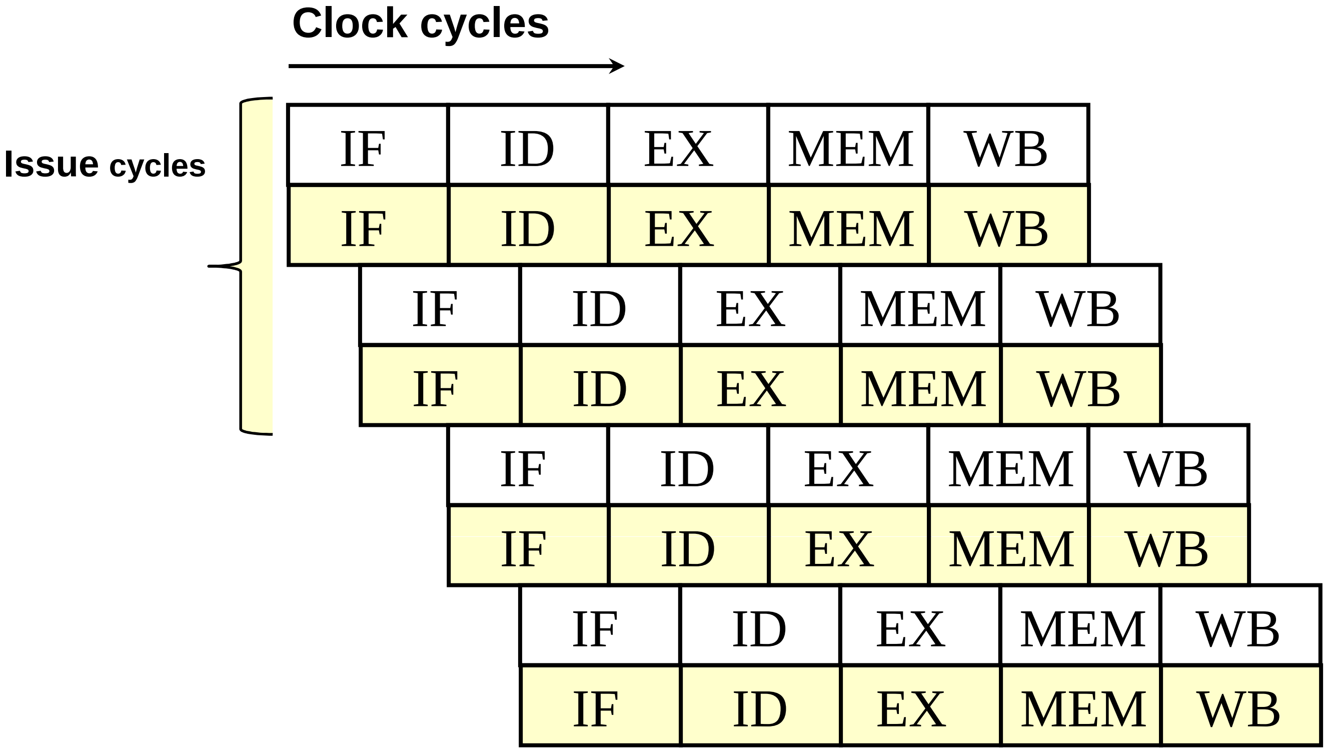
IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

IF ID EX MEM WB

Superpipelined Superscalar Execution



Superscalar Issues

- How to fetch multiple instructions in time (across basic block boundaries) ?
- Predicting branches
- Non-blocking memory system
- Tune #resources(FUs, ports, entries, etc.)
- Handling dependencies
- How to support precise interrupts?
- How to recover from mis-predicted branch path?
- For the latter two issues we need to look at sequential look-ahead and architectural state

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)							
L.D	F2,48(R3)							
MUL.D	F0,F2,F4							
SUB.D	F8,F2,F6							
DIV.D	F10,F0,F6							
ADD.D	F6,F8,F2							
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF						
L.D	F2,48(R3)	IF						
MUL.D	F0,F2,F4							
SUB.D	F8,F2,F6							
DIV.D	F10,F0,F6							
ADD.D	F6,F8,F2							
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX					
L.D	F2,48(R3)	IF	EX					
MUL.D	F0,F2,F4		IF					
SUB.D	F8,F2,F6		IF					
DIV.D	F10,F0,F6							
ADD.D	F6,F8,F2							
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX				
SUB.D	F8,F2,F6		IF	EX				
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX			
SUB.D	F8,F2,F6		IF	EX	EX			
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF				
MUL.D	F12,F2,F4							

stall because of data dep.

cannot be fetched because window full

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX		
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX		
MUL.D	F12,F2,F4					IF		

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				
ADD.D	F6,F8,F2			IF		EX	EX	
MUL.D	F12,F2,F4					IF	cannot execute structural hazard	

Example of Superscalar Processor Execution

- Superscalar processor organization:
  - simple pipeline: IF, EX, WB
  - fetches 2 instructions each cycle
  - 2 ld/st units, dual-ported memory; 2 FP adders; 1 FP multiplier
  - Instruction window (buffer between IF and EX stage) is of size 2
  - FP ld/st takes 1 cc; FP +/- takes 2 cc; FP * takes 4 cc; FP / takes 8 cc

Cycle		1	2	3	4	5	6	7
L.D	F6,32(R2)	IF	EX	WB				
L.D	F2,48(R3)	IF	EX	WB				
MUL.D	F0,F2,F4		IF	EX	EX	EX	EX	WB
SUB.D	F8,F2,F6		IF	EX	EX	WB		
DIV.D	F10,F0,F6			IF				EX
ADD.D	F6,F8,F2			IF		EX	EX	WB
MUL.D	F12,F2,F4					IF		?

Register Renaming

- A technique to eliminate anti- and output dependencies
- Can be implemented
  - by the compiler
    - advantage: low cost
    - disadvantage: “old” codes perform poorly
  - in hardware
    - advantage: binary compatibility
    - disadvantage: extra hardware needed
- We describe general idea

Register Renaming

- there's a *physical register file* larger than *logical register file*
- *mapping table* associates logical registers with physical register
- when an instruction is decoded
  - its physical source registers are obtained from mapping table
  - its physical destination register is obtained from a *free list*
  - mapping table is updated

before: `add r3,r3,4`

mapping table:

r0	R8
r1	R7
r2	R5
r3	R1
r4	R9

free list: R2 R6

after: `add R2,R1,4`

mapping table:

r0	R8
r1	R7
r2	R5
r3	R2
r4	R9

free list: R6

Eliminating False Dependencies

- How register renaming eliminates false dependencies:

- Before:
  - addi r1, r2, 1
  - addi r2, r0, 0
  - addi r1, r0, 1
- After (free list: R7, R8, R9)
  - addi R7, R5, 1
  - addi R8, R0, 0
  - addi R9, R0, 1

Limitations of Multiple-Issue Processors

- Available ILP is limited (we’re not programming with parallelism in mind)
- Hardware cost
  - adding more functional units is easy
  - more memory ports and register ports needed
  - dependency check needs O(n2) comparisons
- Limitations of VLIW processors
  - Loop unrolling increases code size
  - Unfilled slots waste bits
  - Cache miss stalls pipeline
    - Research topic: scheduling loads
  - Binary incompatibility (not EPIC)

Measuring available ILP: How?

- Using existing compiler
- Using trace analysis
  - Track all the real data dependencies (RaWs) of instructions from issue window
    - register dependence
    - memory dependence
  - Check for correct branch prediction
    - if prediction correct continue
    - if wrong, flush schedule and start in next cycle

Trace analysis

Program

```
For i := 0..2
   A[i] := i;
S := X+3;
```

Compiled code

```
        set  r1,0
        set  r2,3
        set  r3,&A
Loop:   st   r1,0(r3)
        add  r1,r1,1
        add  r3,r3,4
        brne r1,r2,Loop
        add  r1,r5,3
```

Trace

```
set  r1,0
set  r2,3
set  r3,&A
st   r1,0(r3)
add  r1,r1,1
add  r3,r3,4
brne r1,r2,Loop
st   r1,0(r3)
add  r1,r1,1
add  r3,r3,4
brne r1,r2,Loop
st   r1,0(r3)
add  r1,r1,1
add  r3,r3,4
brne r1,r2,Loop
add  r1,r5,3
```

How parallel can this code be executed?

Trace analysis

Parallel Trace

```
set  r1,0      set  r2,3      set  r3,&A
st   r1,0(r3)  add  r1,r1,1   add  r3,r3,4
st   r1,0(r3)  add  r1,r1,1   add  r3,r3,4   brne r1,r2,Loop
st   r1,0(r3)  add  r1,r1,1   add  r3,r3,4   brne r1,r2,Loop
brne r1,r2,Loop
add  r1,r5,3
```

Max ILP =

$$\text{Speedup} = L_{\text{parallel}} / L_{\text{serial}} = 16 / 6 = 2.7$$

Is this the maximum?

Ideal Processor

Assumptions for ideal/perfect processor:

1. *Register renaming* – infinite number of virtual registers => all register WAW & WAR hazards avoided
2. *Branch and Jump prediction* – Perfect => all program instructions available for execution
3. *Memory-address alias analysis* – addresses are known. A store can be moved before a load provided addresses not equal

Also:

- unlimited number of instructions issued/cycle (unlimited resources), and
- unlimited instruction window
- perfect caches
- 1 cycle latency for all instructions (FP *,/)

Programs were compiled using MIPS compiler with maximum optimization level