Instruction Level Parallelism

Pipeline Hazards and LL Pipeline

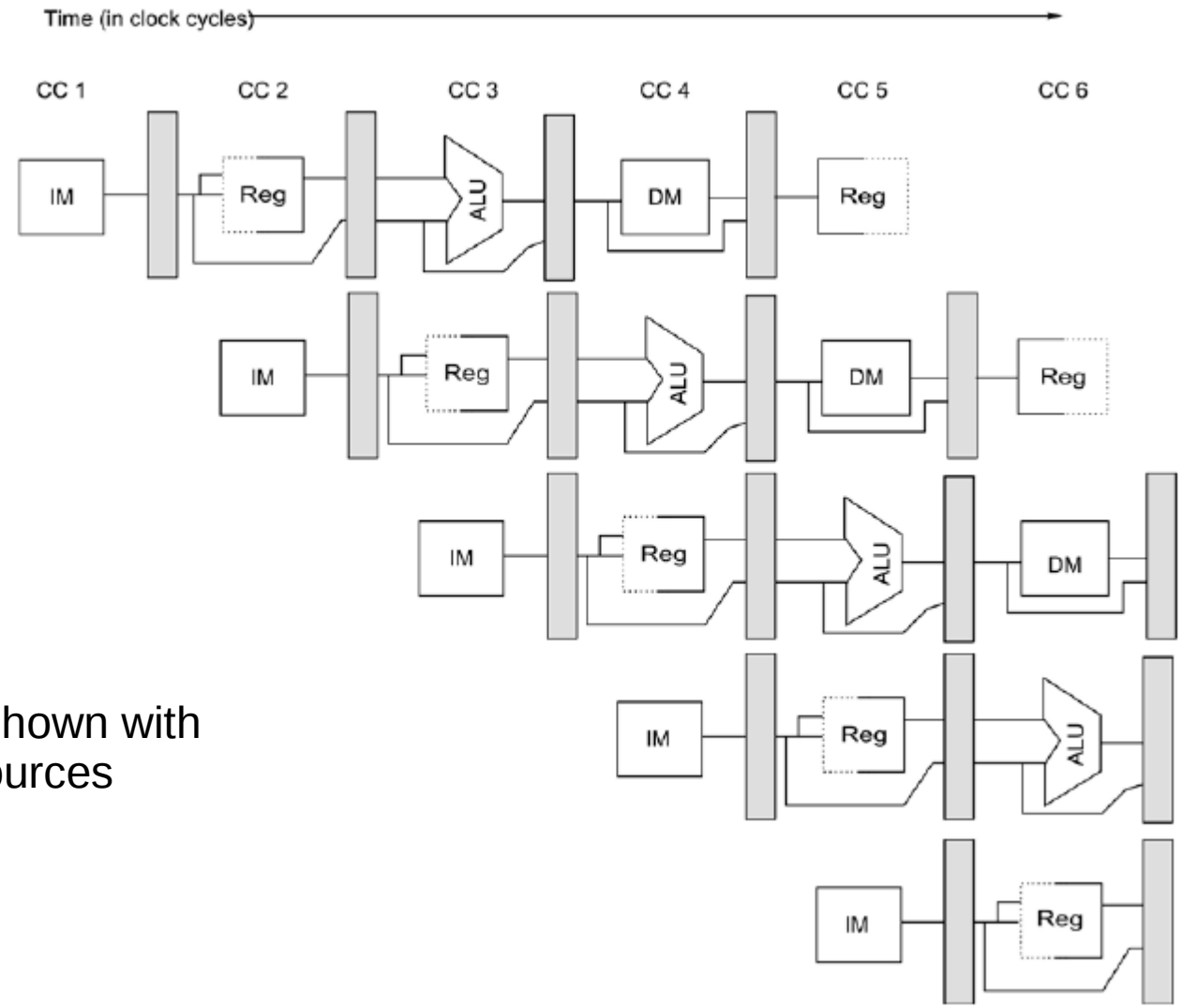
Indian Institute of Technology Tirupati

Jaynarayan T Tudu
[jtt@iittp.ac.in]

Computer System Architecture (CS5202)
19th March, 2020

Pipeline Architecture

Review of the basic pipeline architecture



The stages are being shown with the corresponding resources

Program Execution Scenario

DADD R1 R2 R3

DSUB R4 R1 R5

AND R6 R1 R7

OR R8 R1 R9

XOR R10 R1 R11

BEQ R1 R4 offset

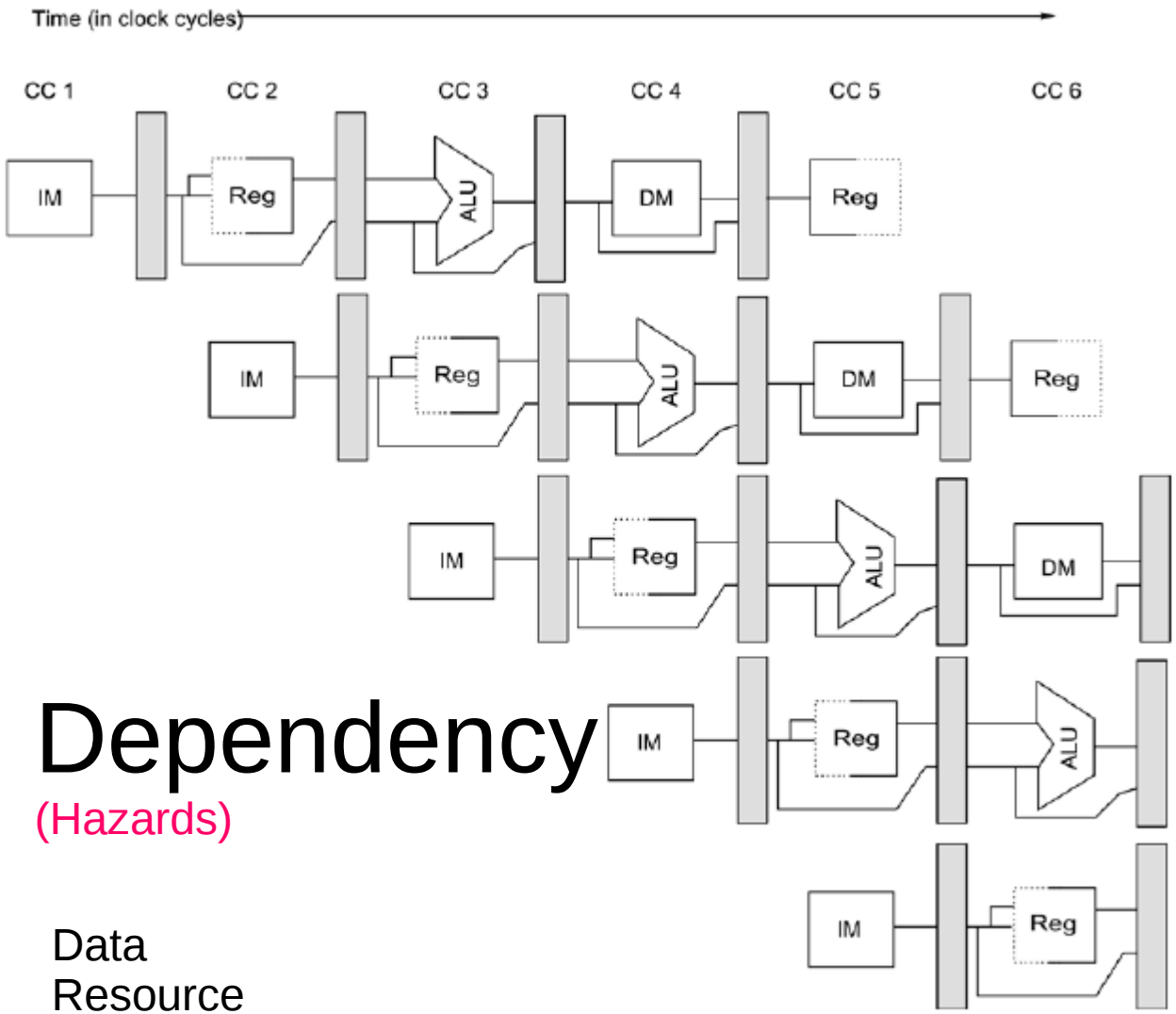
XOR R1 R4 R11

NOR R10 R4 R11

JMP 100(offset)

…….

…….



Dependency

(Hazards)

Data

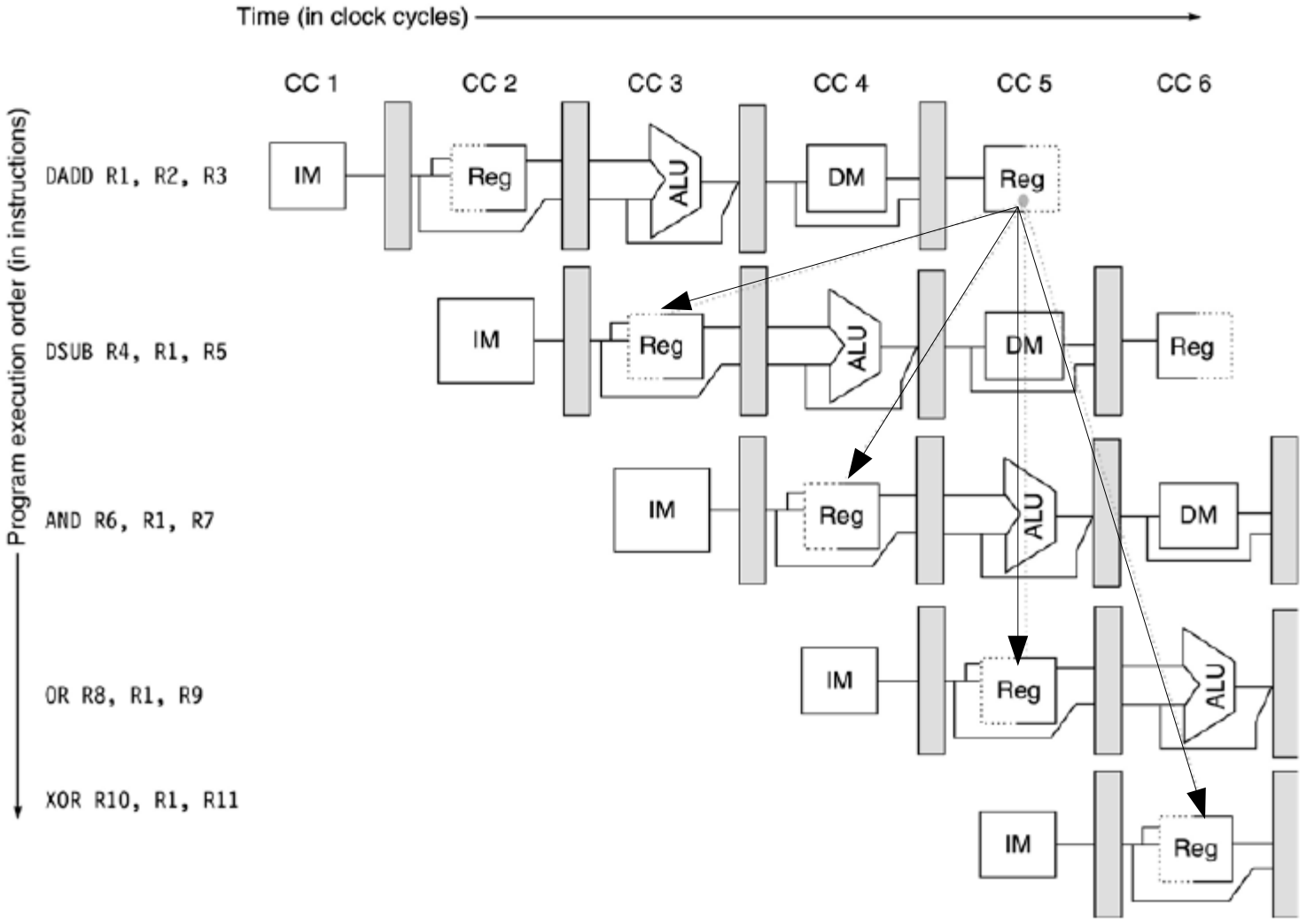
Resource

Control

Data Dependency

The challenges of data sharing (Hazards)

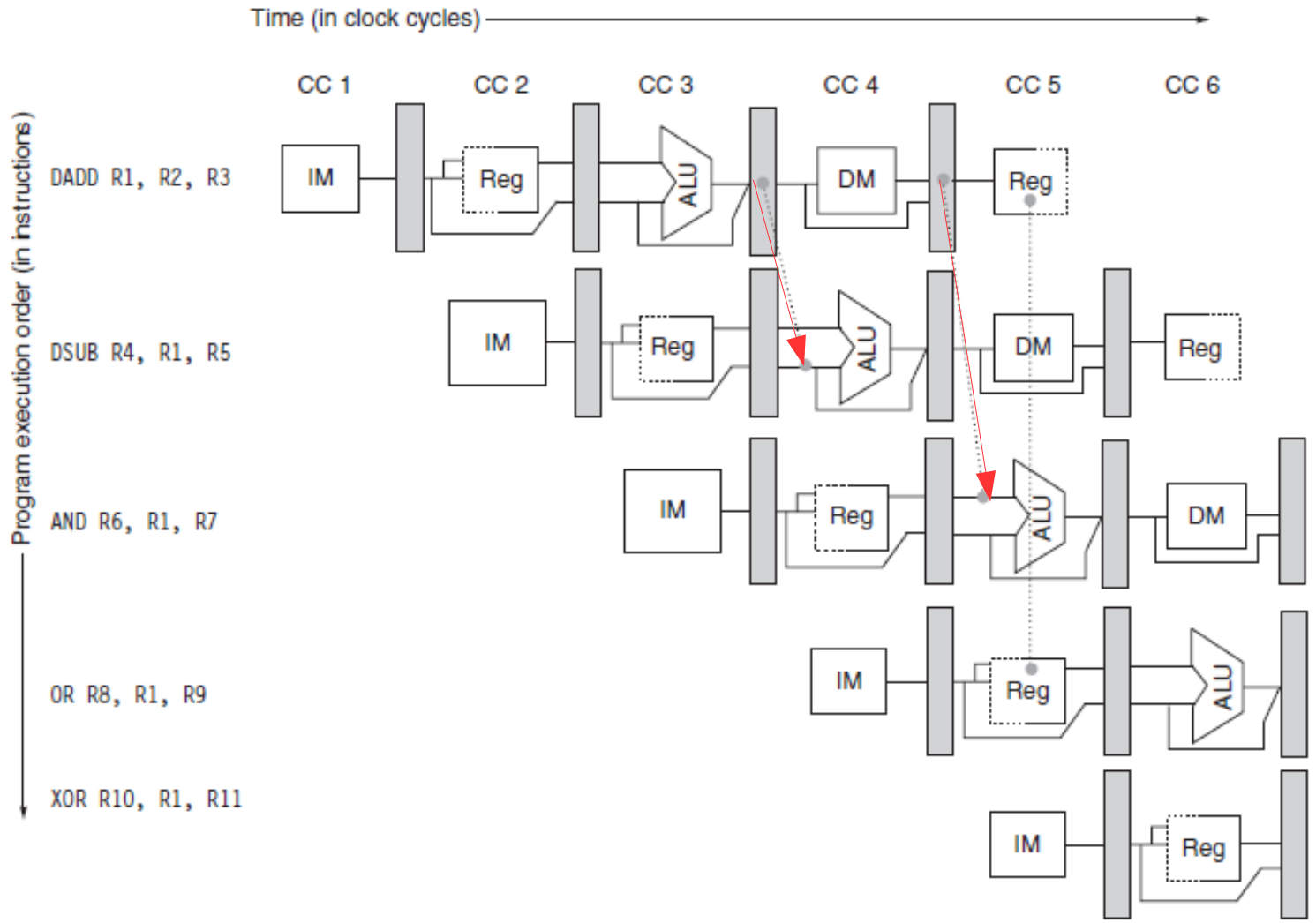
Data Hazards
R A W
W A R
R A R
W A W



Data Dependency: Solutions

Data Forwarding / Bypassing /Short circuiting

Data Hazards
R A W
W A R
R A R
W A W



More solutions will be discussed in detail during LL pipeline and superscalar!

Resource Dependency

DADD R1 R2 R3

DSUB R4 R1 R5

AND R6 R1 R7

OR R8 R1 R9

XOR R10 R1 R11

BEQ R1 R4 offset

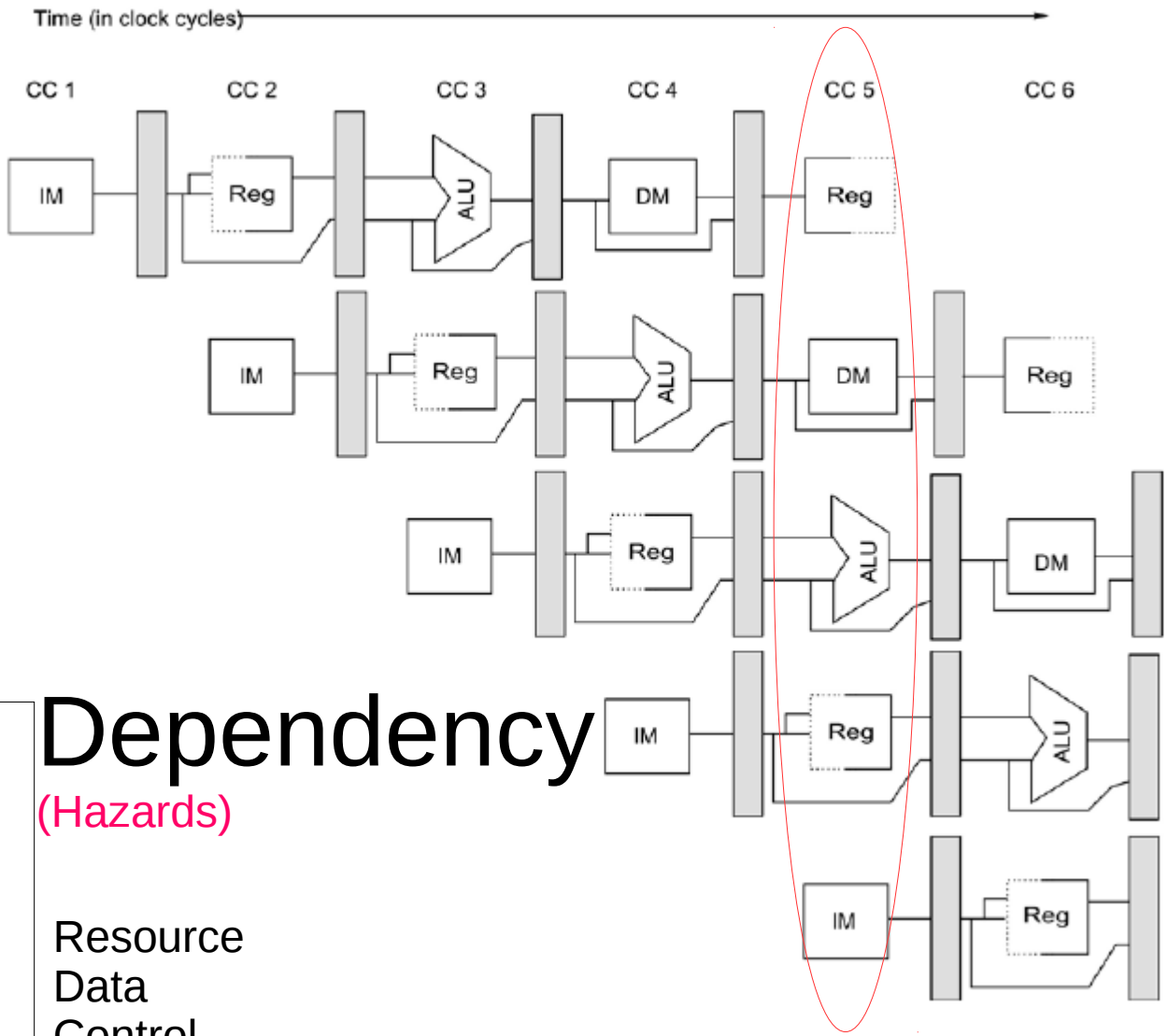
XOR R1 R4 R11

NOR R10 R4 R11

JMP 100(offset)

…….

…….



Dependency

(Hazards)

Resource

Data

Control

Resource Dependency

DADD R1 R2 R3

DSUB R4 R1 R5

AND R6 R1 R7

OR R8 R1 R9

XOR R10 R1 R11

BEQ R1 R4 offset

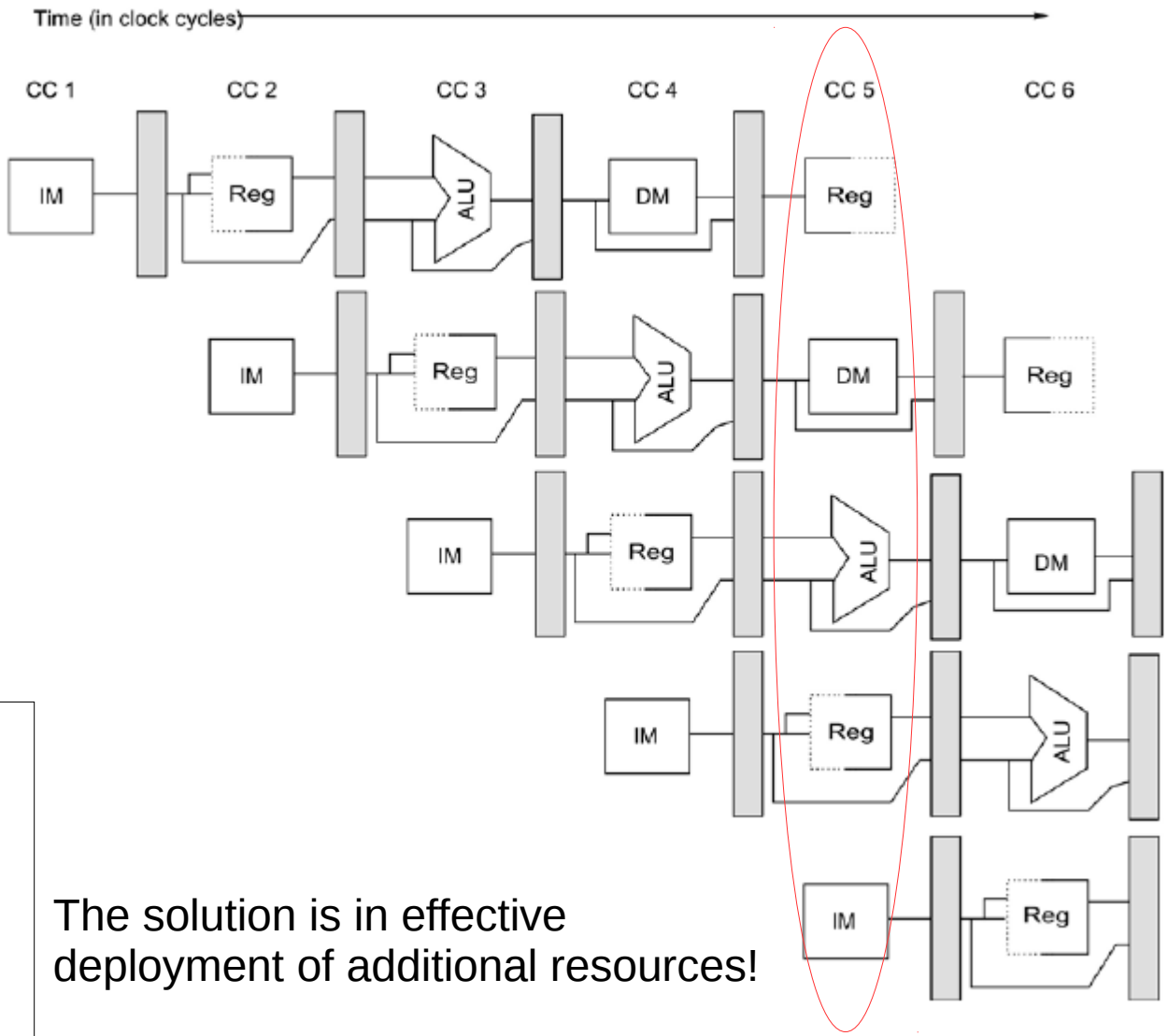
XOR R1 R4 R11

NOR R10 R4 R11

JMP 100(offset)

…….

…….



The solution is in effective deployment of additional resources!

IM and DM, additional ports to Reg etc

Control Dependency

PC = 00: DADD R1 R2 R3

PC = 04: DSUB R4 R1 R5

PC = 08: AND R6 R1 R7

PC = 16: OR R8 R1 R9

PC = 20: XOR R10 R1 R11

PC = 24: BEQ R1 R4 offset

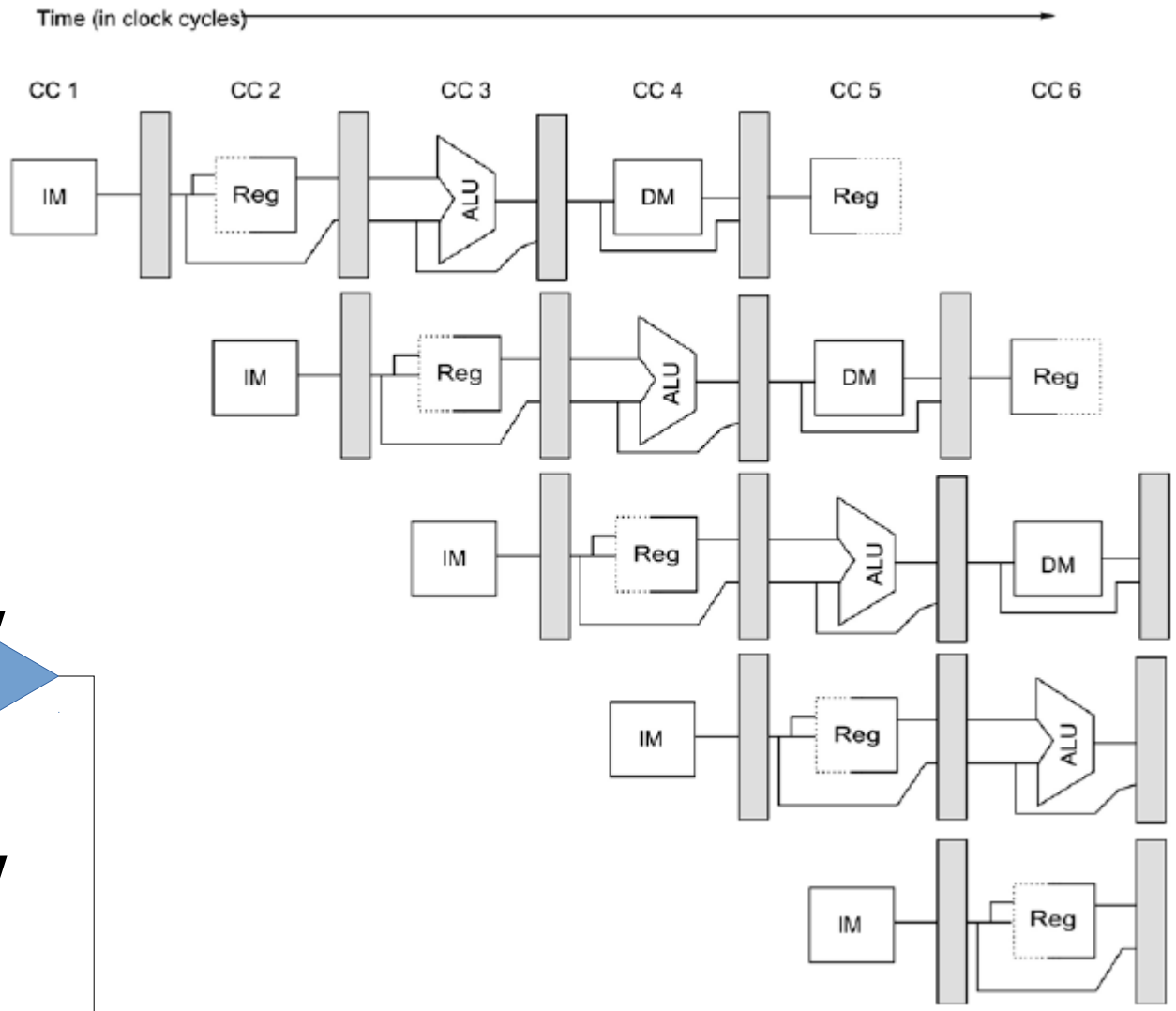
PC = 28: XOR R1 R4 R11

PC = 32: NOR R10 R4 R11

PC = 36: JMP 100(offset)

…….

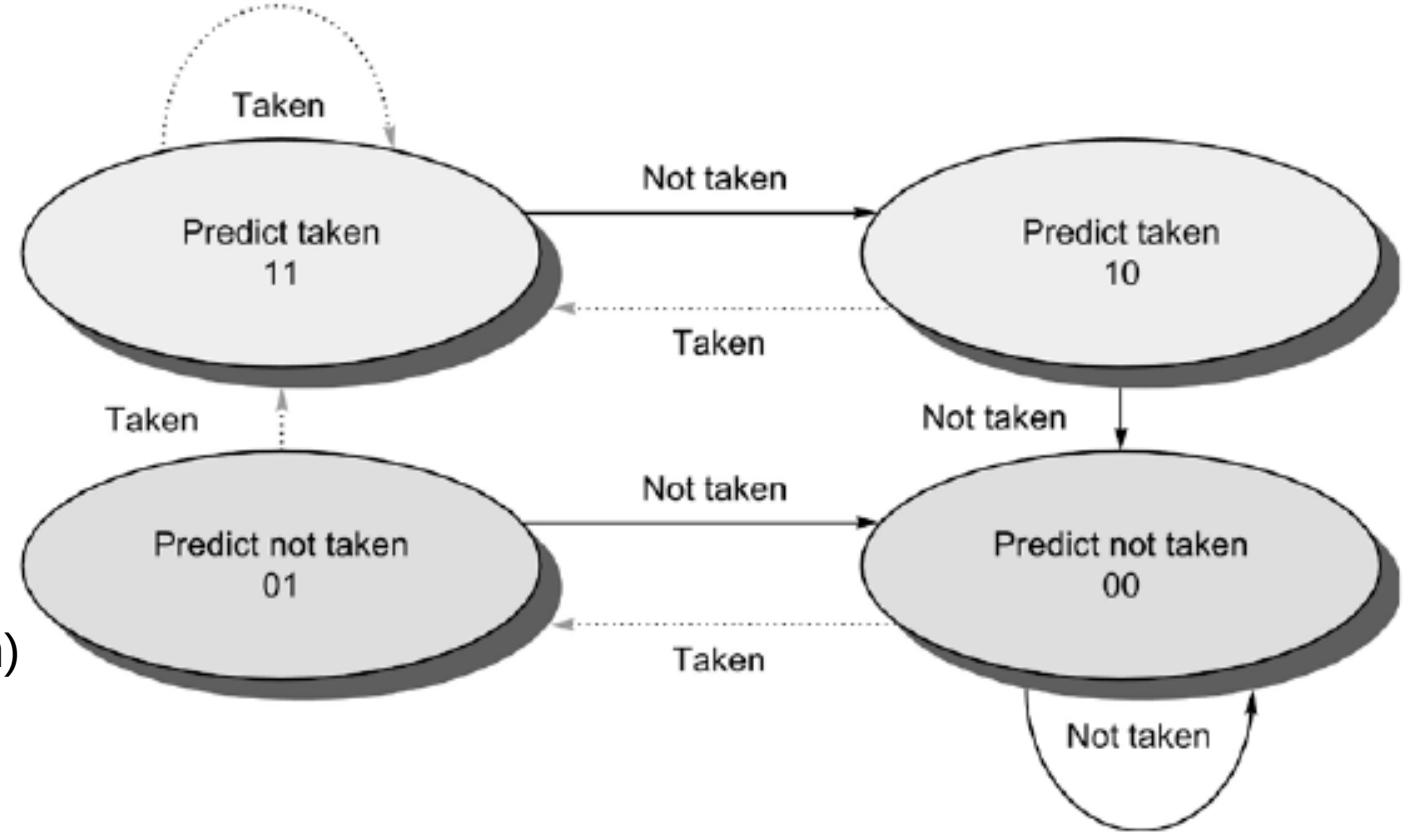
PC = 24 + offset: …….



Branch Prediction: Control Dependency

The control hazards (Causes the break of normal pipeline flow)

Predict the control path (branch prediction)



** Advanced branch predictors will be covered during Super-scalar architecture*

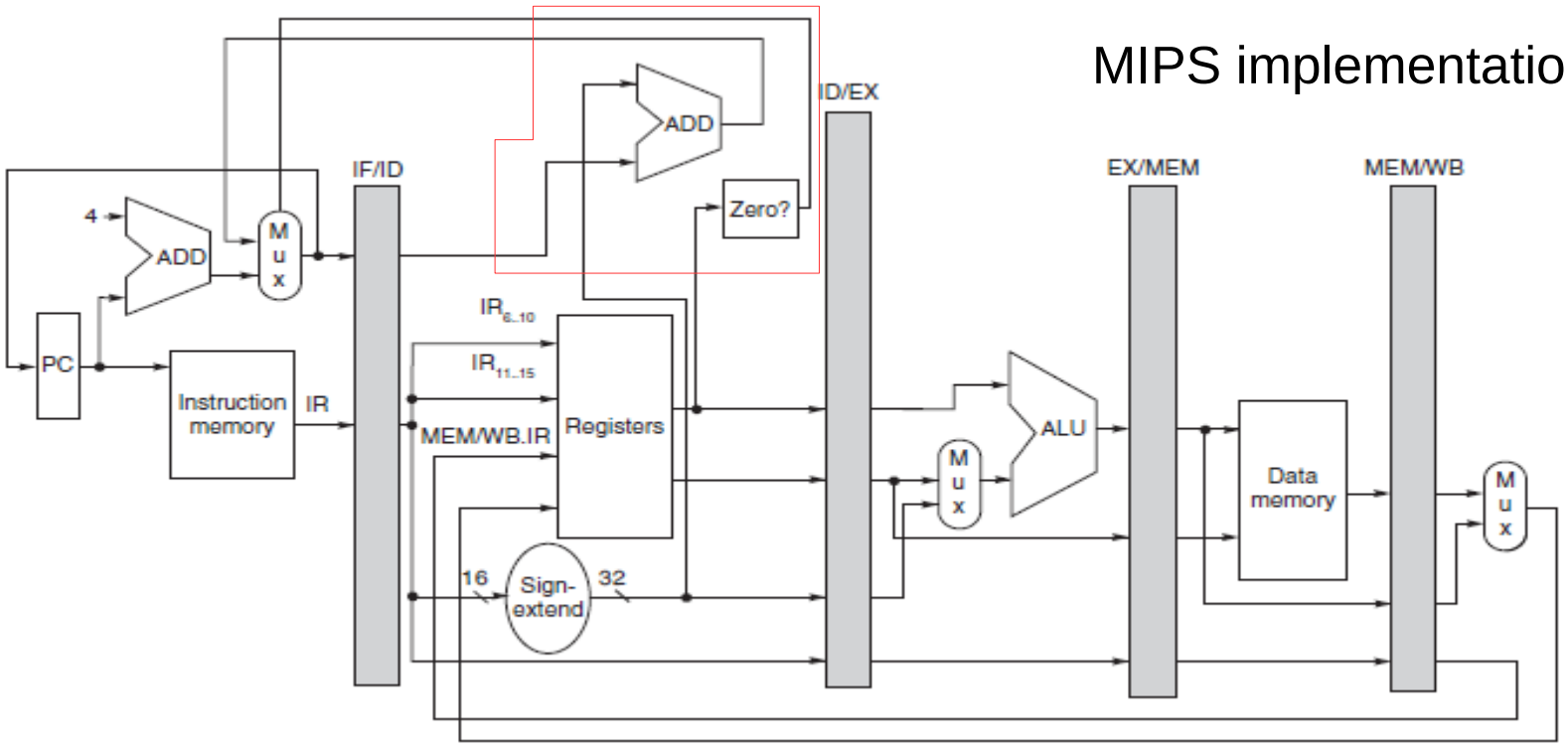
Branch Prediction: Control Dependency

This requires two things:

- Branching decision (taken or Not taken)
- Branch target address (effective address)

Both have been moved to ID,
Just we don't want to wait!

MIPS implementation



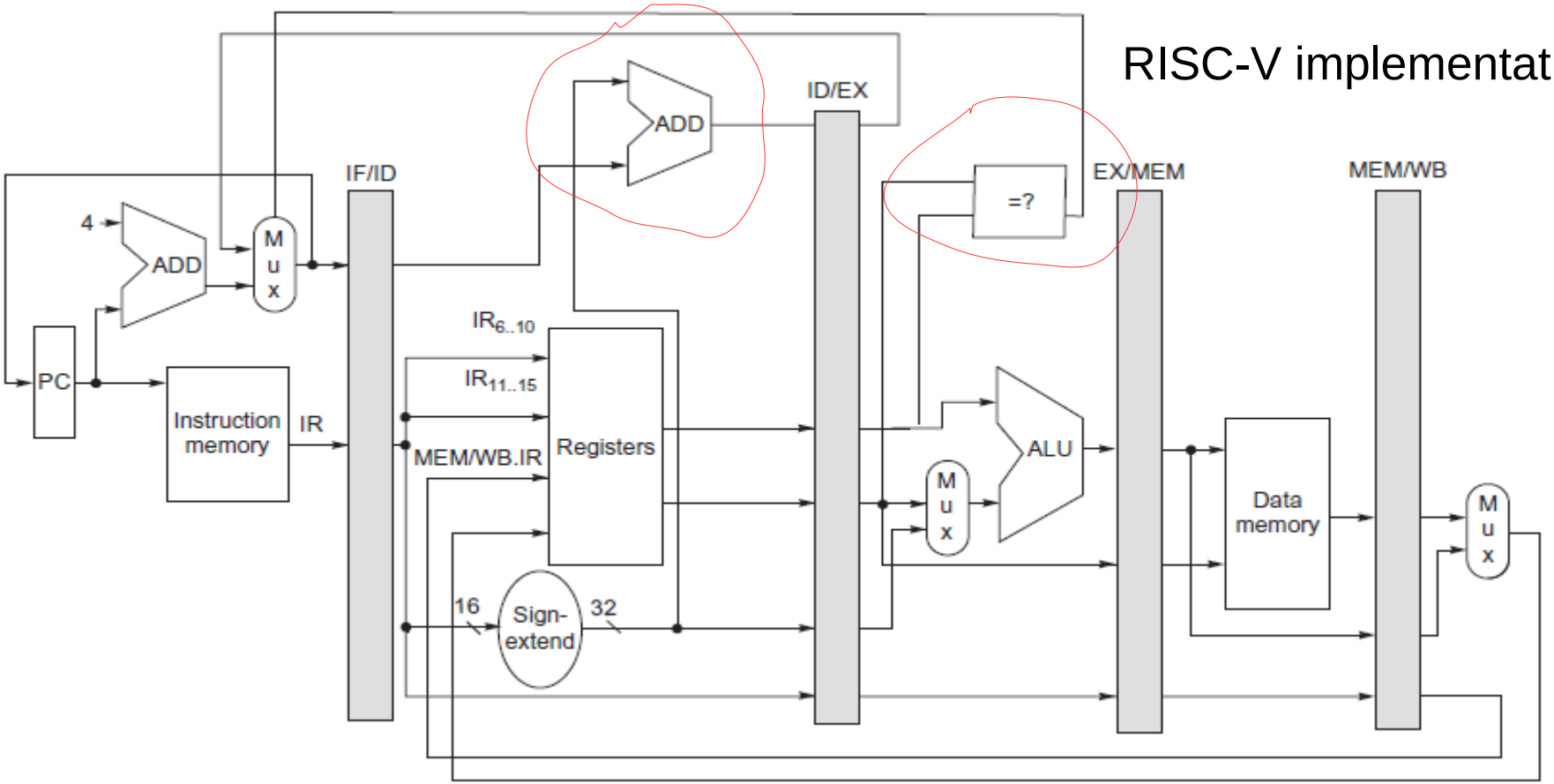
** Advanced branch predictors will be covered during Super-scalar architecture*

Branch Prediction: Control Dependency

This requires two things:

- Branching decision (taken or Not taken)
- Branch target address (effective address)

RISC-V implementation



** Advanced branch predictors will be covered during Super-scalar architecture*

Performance with Hazards

Hazards are causing pipeline to stall – extra cycle penalty!

In ideal pipeline: cycle per instruction = 1.

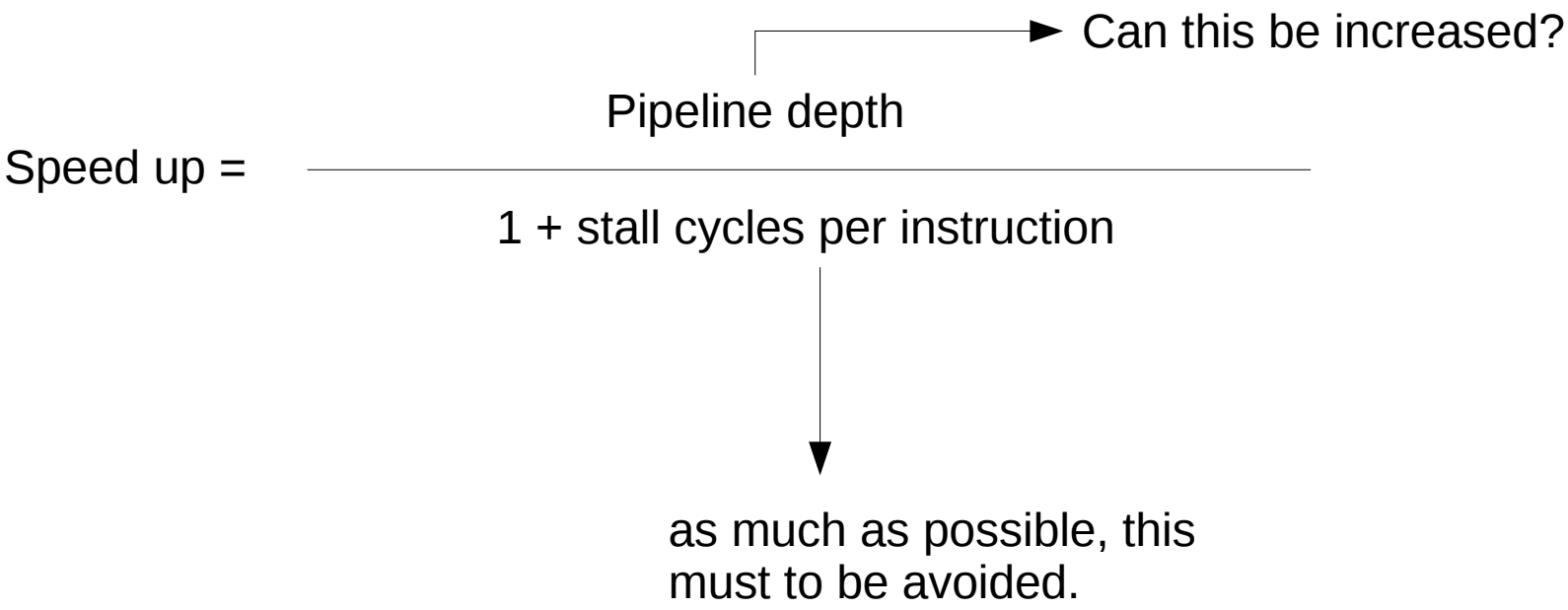
Therefore, with hazards, the cycle per instruction = 1 + stall cycles per instruction

$$\text{Speed up} = \frac{\text{CPI in unpipelined (generally multicycle)}}{1 + \text{stall cycles per instruction}}$$

The above equation could be expressed also as follow:

$$\text{Speed up} = \frac{\text{Pipeline depth}}{1 + \text{stall cycles per instruction}}$$

Deep Pipeline Architecture



The question is how to increase pipeline depth without increasing stall cycles?

Analyzing the Execution stage!

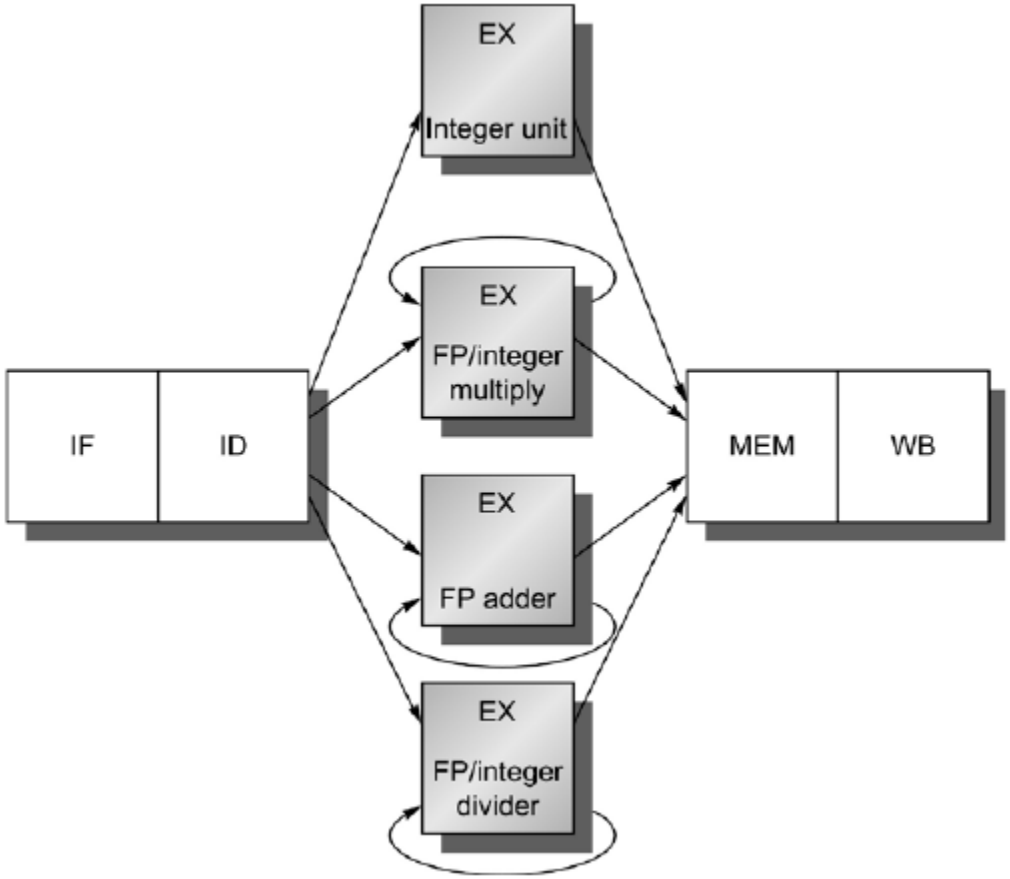
Long Latency Pipeline

Multiple units in EX stage

Integer Unit
Floating point/Integer multiply
FP Adder
FP/Integer Divider

Assumption:

Each of the execution unit is non-pipeline.



How to decide on which unit or where do a deeper pipeline is needed?

FP Pipeline Architecture

To decide which unit to be pipelined for performance gain, we use the following specification.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Latency: number of cycles between the production and the consumption of the results (it helps designer to decide on reducing stalls).

Initiation interval: elapsed number of cycle between issuing of two operations of a given type
(For pipeline this would be always 1, for multi-cycle it may not be)

.

FP Pipeline Architecture

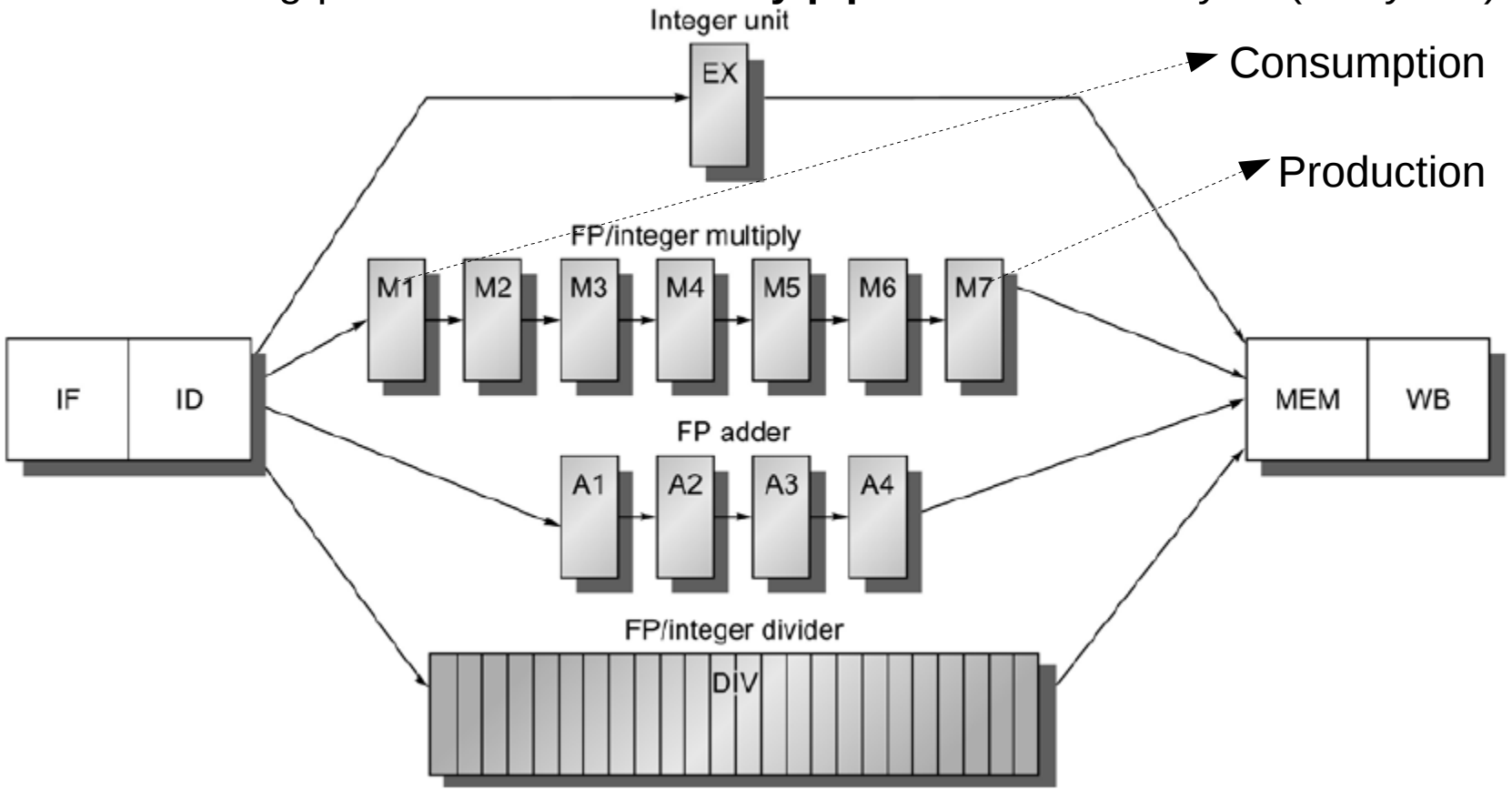
Block diagram of deeper pipeline based on the given specification.

Integer ALU: only **one** stage

FP/Integer Multiply: fully pipelined with **seven** stages.

FP adder: fully pipelined with **four** stages.

Floating-point division is **not fully pipelined** but multicycle (24 cycles).



FP Pipeline Architecture

Latency and Initiation interval for each of the pipeline units. Note that in case of FP divider the initiation interval is 25 instead of 1.

Functional unit	Latency	Initiation interval
Integer ALU	0 (= 1 - 1)	1
Data memory (integer and FP loads)	1 (= 2 - 1)	1
FP add	3 (= 4 - 1)	1
FP multiply (also integer multiply)	6 (= 7 - 1)	1
FP divide (also integer divide)	24 (= 25 - 1)	25

Latency: number of cycles between production of results and the consumption of results

Initiation interval: elapsed number of cycles between issuing of two operations/instructions of a given type.

FP Pipeline Architecture: Timing

Timing of independent set of instructions.

Progress of time (clock cycle)

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
fadd.d			IF	ID	*EX*	**MEM**	WB				
fsd				IF	ID	*EX*	**MEM**	WB			

Data is required at this stage

Results are available at this stages

Figure is from 6th edition of the text book, however, for your reading you may take 5th edition, There are some printing bugs in 6th edition.

FP Pipeline Architecture: Hazards

Hazards and forwarding in long latency pipeline:

Data dependency (Consumer must get the updated data)
Control (Unpredictable control path)
Structural hazards (No two stages can access a single resources at a time)

1) Because the divide unit is non-pipeline, structural hazard can occur. This needs to be detected and the issuing instructions need to be stalled.

2) Due to varying running time of each of the instruction, the number of register writes required in a cycle is more than 1.

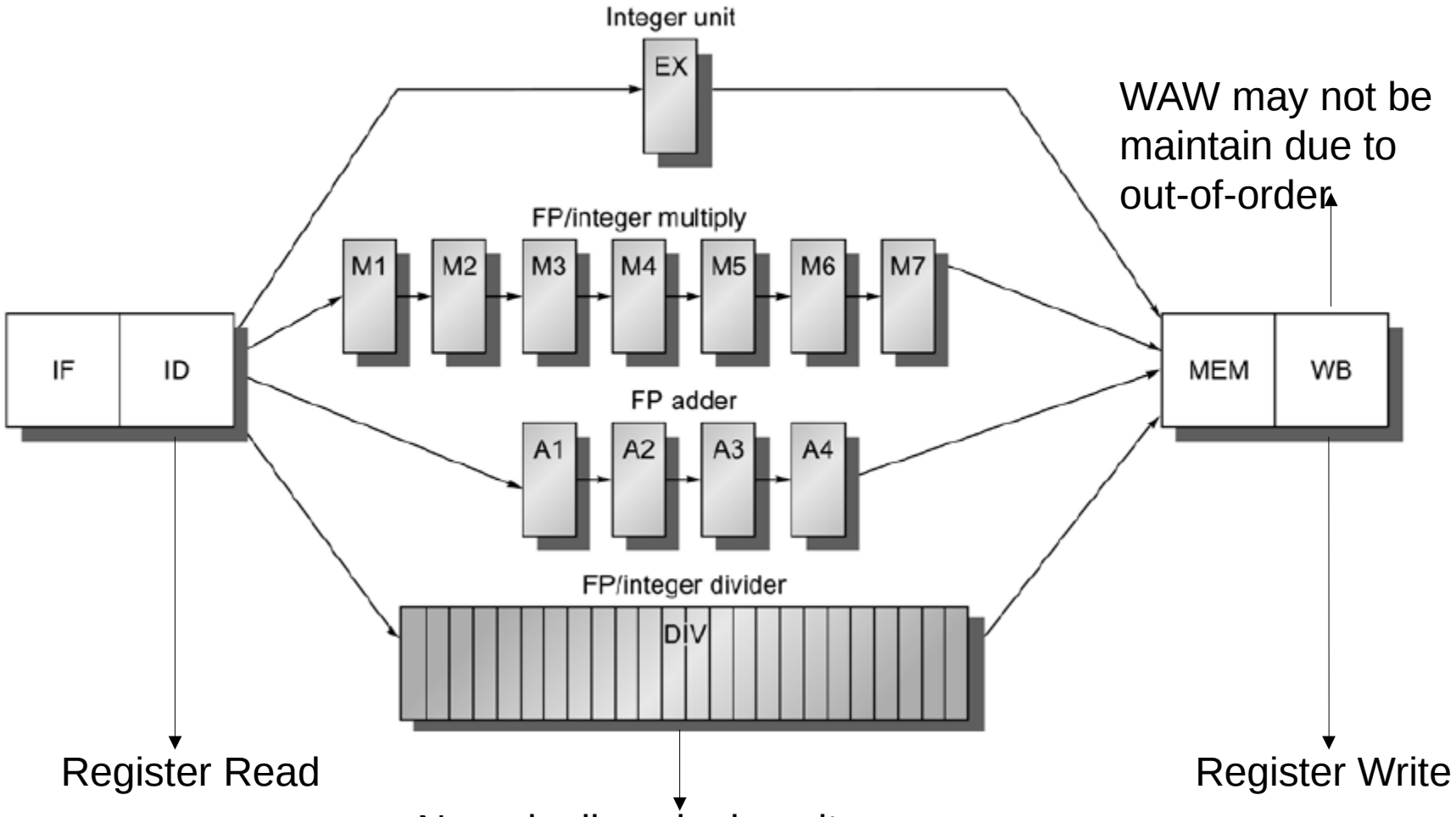
3) WAW hazards are possible, because instruction no longer reach WB in order. Note that WAR hazards shall never occur since the read happens in ID stage and the write at WB stage.

4) Instruction can complete in different order that they were issued (issued in order but completed in out-of-order), this may leads to imprecise exception handling.

5) Because of longer latency of operations, the stalls due to RAW hazards will be more frequent.

FP Pipeline Architecture: Hazards

Hazards and forwarding in long latency pipeline:



FP Pipeline Architecture: Hazards

Pipeline stalls due to data dependency hazards.

Solution: stall and forwarding

	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
fld f4,0(x2)	IF	ID	EX	MEM	WB												
fmul.d f0,f4,f6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
fadd.d f2,f0,f8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
fsd f2,0(x2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

f4, f0, f2 are the floating point registers leading to RAW dependency.

RAW dependency

RAW dependency

Due to structural hazards

FP Pipeline Architecture: Solutions

Solutions to hazards.

Two data dependency hazards: RAW and WAW are of interest
And the structural hazards.

Structural hazards

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

WAW

Situation for WAW to cause issue: if fld f2, 0(x2) would have been issued a cycle before. fld f2, 0(x2) and fadd.d f2, f4, f6 would cause WAW.

FP Pipeline Architecture: Solutions

Solutions to hazards.

Structural hazards due to WB and MEM:

- detect hazards and stall
- detection can be done at ID stage or at MEM stage
- stall the issue at ID stage or stall before entering to MEM or WB

Solving the WAW hazards:

- delay the issue of fld instruction unitl fadd.d enters to MEM stage
- stop fadd.d to write back its result, and then issue the fld as usual.
- WAW is very rare situation in code.

Hazards among FP and Integer units:

- Hazards can occur among FP instructions or between FP and Integer instruction.
- Having separate register file for FP and Integer unit is a good solution
- Detection of hazards among FP
    - Check for structural hazards
    - Check for RAW hazard
    - Check for WAW hazards

Pipeline Architecture: Exception

The problem due to longer pipeline and out-of-order completion:

DIV.D F0, F2, F4
ADD.D F10, F10, F8
SUB.D F12, F12, F14 ← Exception or interrupts

Where should program returns after handling exception?
To DIV.D or to SUB.D.

How to maintain precise program state?

Can’t afford to lose value of any register!

Pipeline Architecture: Exception

Different situations which are Exception!

- I/O device request
- System Call (invoking OS kernel from user mode)
- Tracing instruction execution
- Break points (like gdb break)
- Integer arithmetic overflow (when a number can’t be represented)
- FP arithmetic anomaly (such as NaN)
- Divide by zero
- Page fault (OS intervention is called for)
- Misaligned memory access (you have to do memory alignment)
- Memory protection violation
- Using an undefine instruction
- Hardware malfunction
- Power failure, Reset, Restart etc.

Beyond Pipeline

Question of interest:

How to increase IPC? IPC >= 1

or

CPI <= 1.0

Limits of Pipeline

Processor Performance revisit:

Performance = 1/CPU_time

CPU_time = time/program

= Instruction/Program x Cycle/Instruction x Time/Cycle

Speed up = Performance of new / Performance of Old

= CPU time in Old / CPU time in new

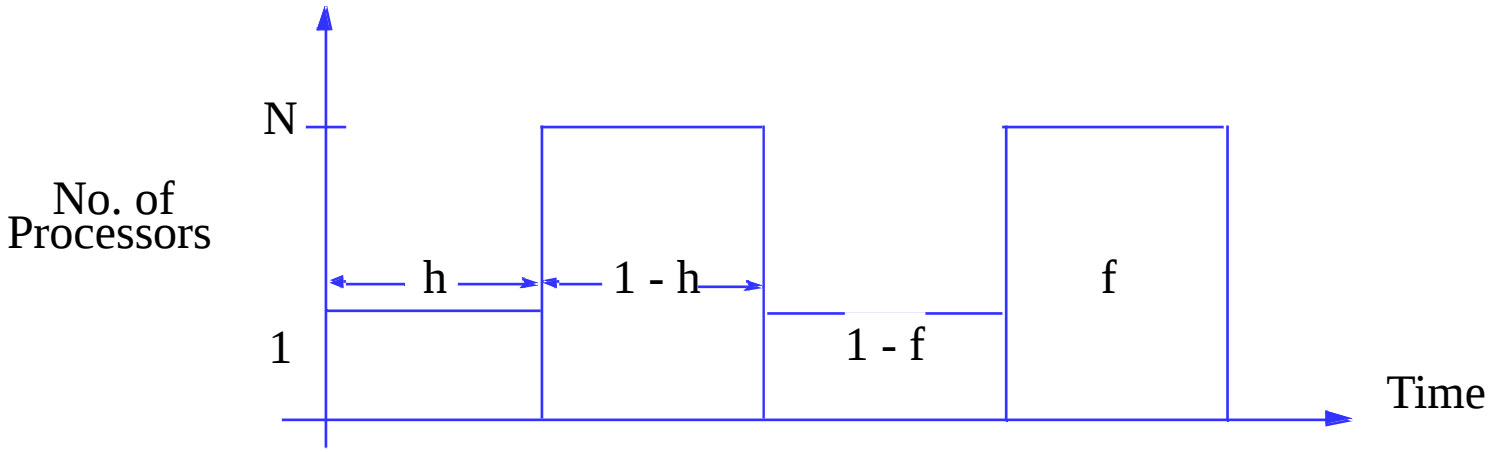
- In the 1980's (decade of pipelining):
  - CPI: between 5.0 to 1.15
- In the 1990's (decade of superscalar):
  - CPI: between 1.15 to 0.5 (best case)
- In the 2000's (decade of multicore):
  - Focus on thread-level parallelism, CPI near to 0.33 (best case)

Limits of Pipeline

Amdhal's Law

Speed up = P1/P2

P1: Performance for entire task using the enhancement
P2: Performance for entire task without enhancement

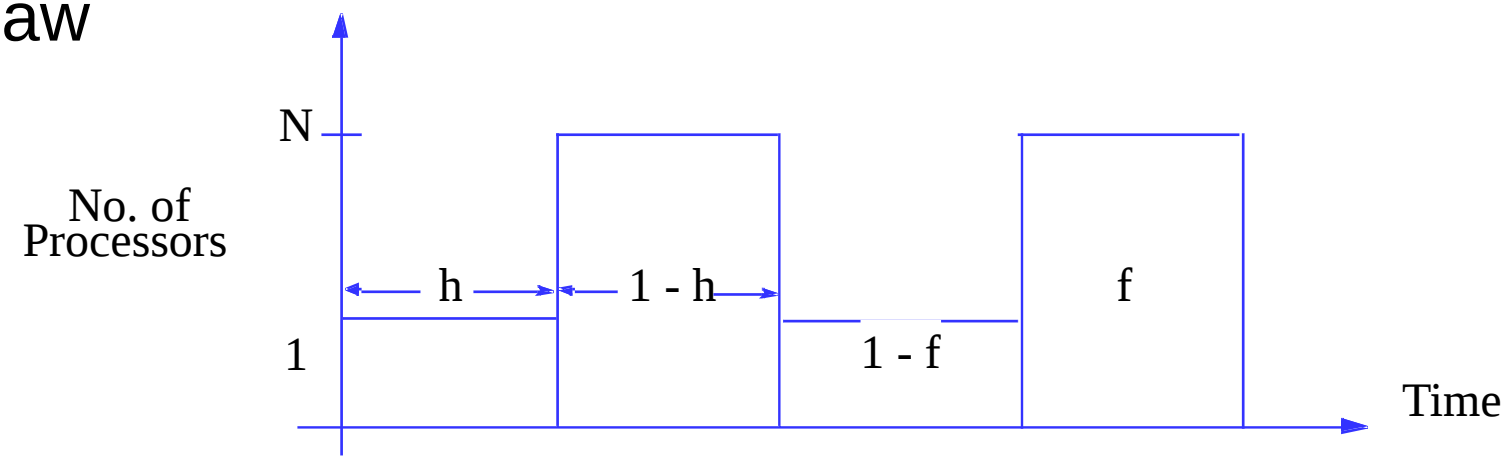


- h = fraction of time in serial code
- f = fraction that is vectorizable
- v = speedup for f
- Overall speedup:

$$Speedup = \frac{1}{1 - f + \frac{f}{v}}$$

Limits of Pipeline

Amdhal's Law

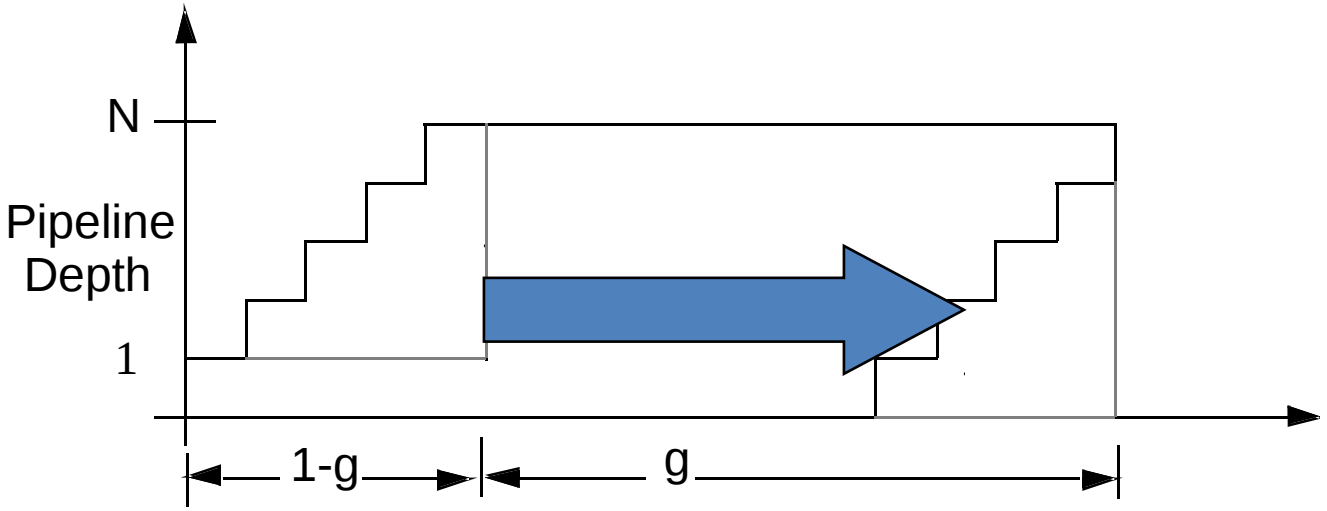


- Sequential bottle neck
- Even if v is infinite, the performance is limited by non-vectorizable code i.e 1-f

$$\lim_{v \to \infty} \frac{1}{1 - f + \frac{f}{v}} = \frac{1}{1 - f}$$

Limits of Pipeline

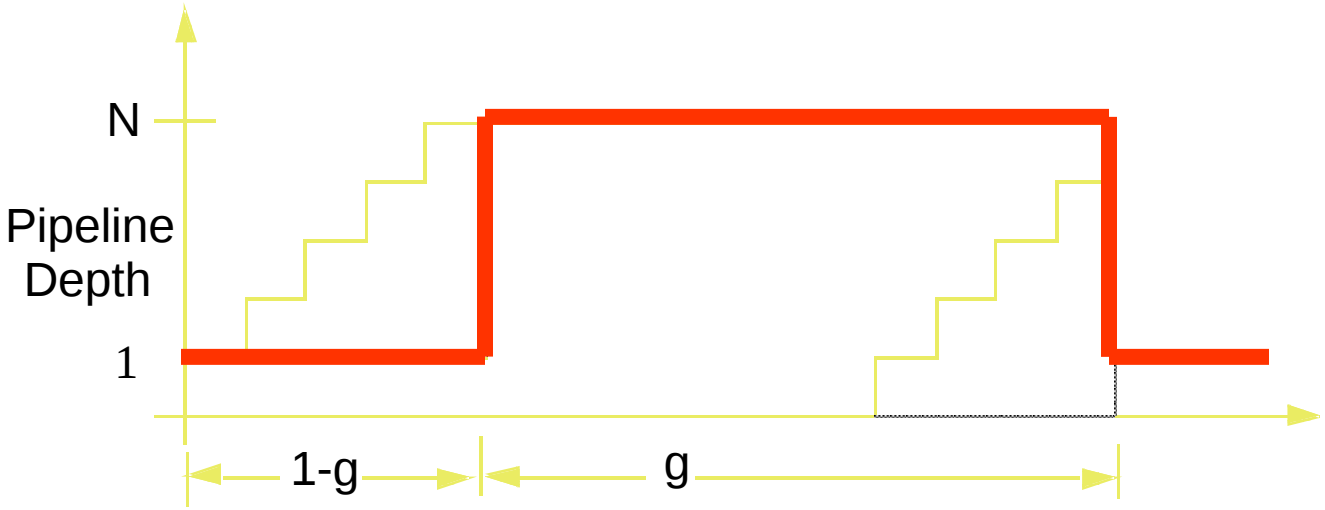
Pipeline Performance Model:



- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

Limits of Pipeline

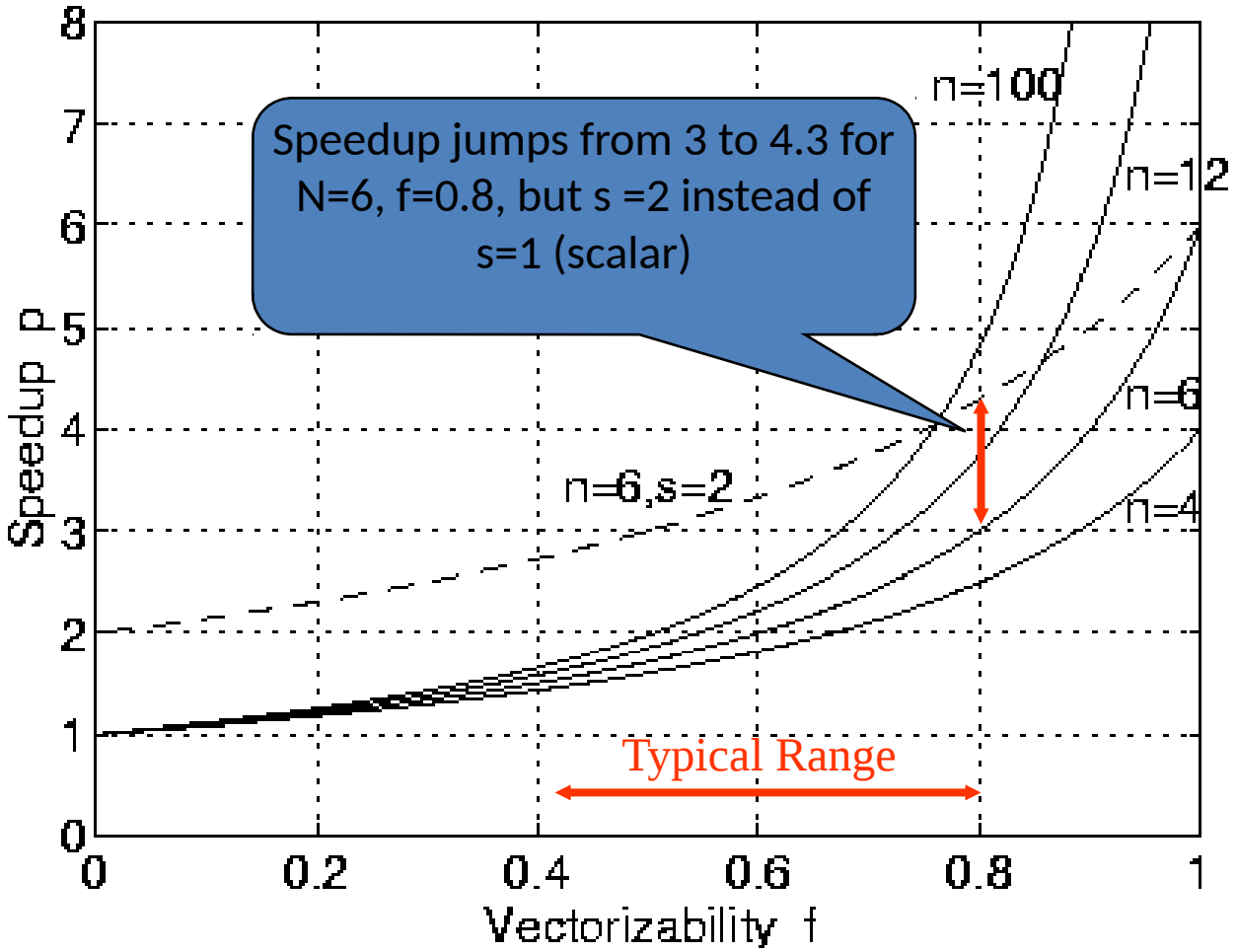
Pipeline Performance Model:



- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)

Beyond Scalar Pipeline

n = pipeline stages
f = fraction of
Vectorizable code
s = scalar size
(for base pipeline
s = 1)



Limits of Pipeline

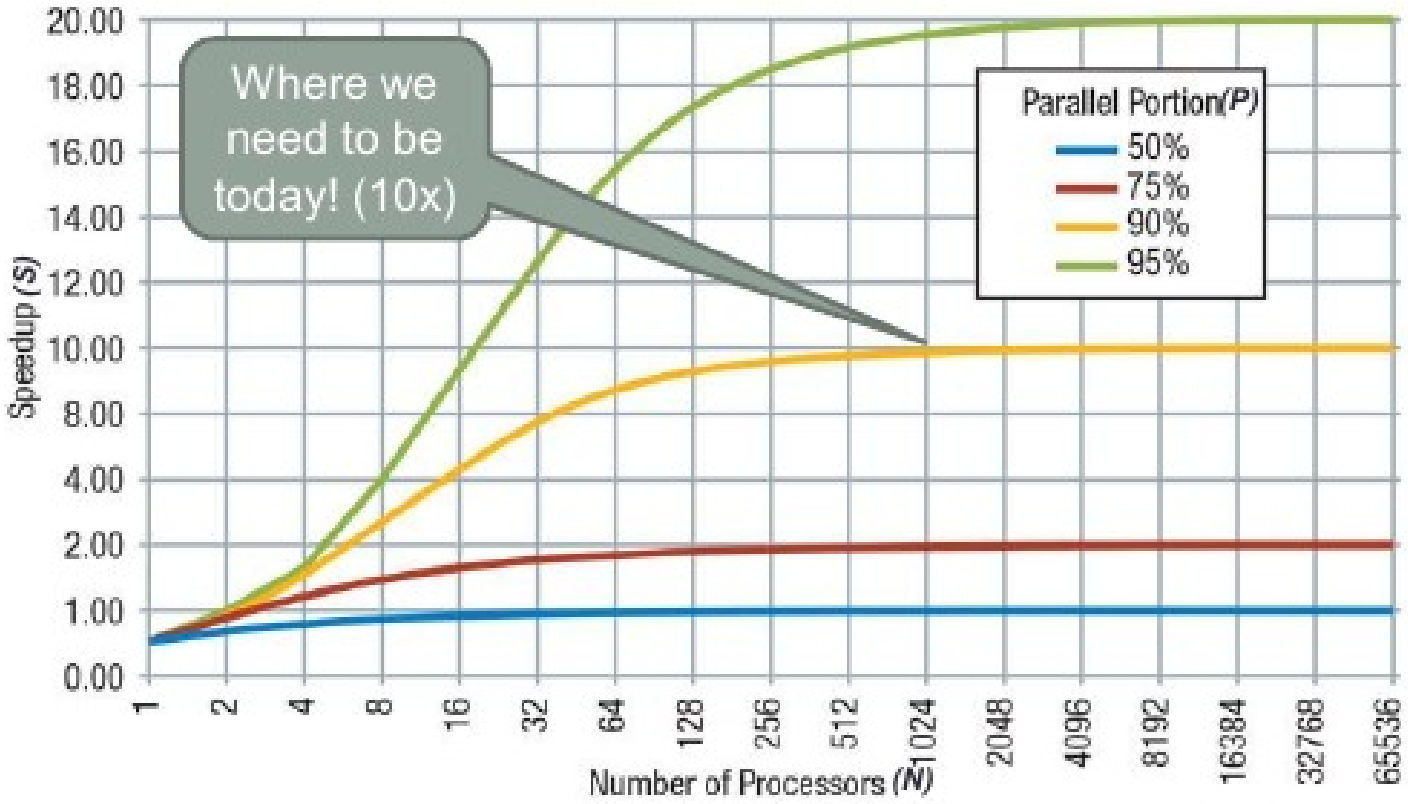
$$\text{Speedup}(N) = \frac{1}{(1-f) + f/N}$$

f - fraction vectorizable
N - number of processors

The challenge of

Amdhal's Law

Look at the
90%
and
95%



Limits of Pipeline

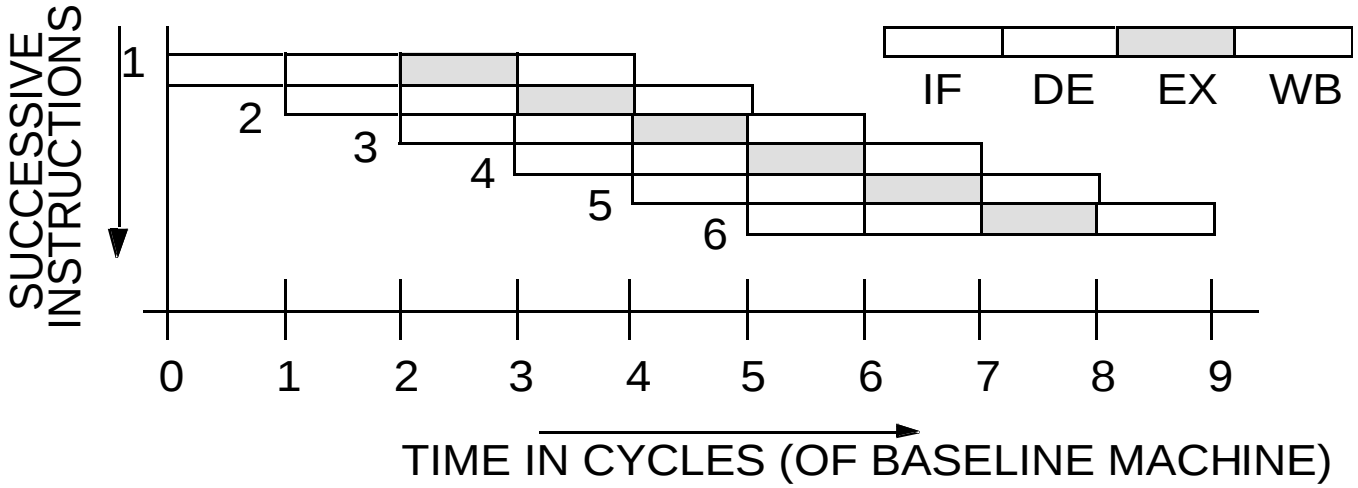
- IBM RISC Experience
  - Control and data dependencies add 15%
  - Best case CPI of 1.15, IPC of 0.87
  - Deeper pipelines (higher frequency) magnify dependence penalties

- This analysis assumes 100% cache hit rates
  - Hit rates approach 100% for some programs
  - Many important programs have much worse hit rates

Classifying ILP Machines

Baseline scalar RISC:

- Issue parallelism = IP = 1 [only one instruction]
- Operation latency = OP = 1
- Peak IPC = 1

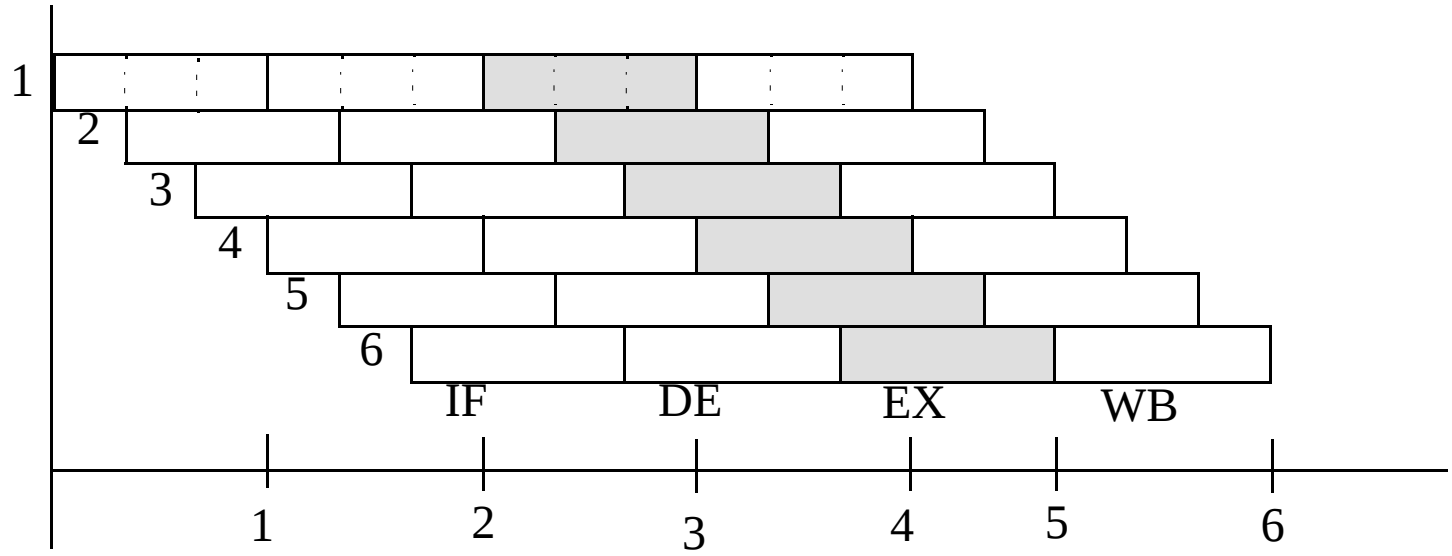


- Jouppi, WRL Reserch Report 89/7, 1989

Classifying ILP Machines

Super-pipelined:

- Cycle time (minor cycle) = *1/m* of baseline
- Issue parallelism = IP = 1 inst/minor cycle
- Operation latency = OP = *m* minor cycles
- Peak IPC = m instr / major cycle (m x speedup?)



- Jouppi, WRL Reserch Report 89/7, 1989

Limits on Instruction Level Parallelism (ILP)

Ideas	IPC achieved
Weiss and Smith [1984]	1.58
Sohi and Vajapeyam [1987]	1.81
Tjaden and Flynn [1970]	1.86 (Flynn's bottleneck)
Tjaden and Flynn [1973]	1.96
Uht [1986]	2.00
Smith et al. [1989]	2.00
Jouppi and Wall [1988]	2.40
Johnson [1991]	2.50
Acosta et al. [1986]	2.79
Wedig [1982]	3.00
Butler et al. [1991]	5.8
Melvin and Patt [1991]	6
Wall [1991]	7 (Jouppi disagreed)
Kuck et al. [1972]	8
Riseman and Foster [1972]	51 (no control dependences)
Nicolau and Fisher [1984]	90 (Fisher's optimism)

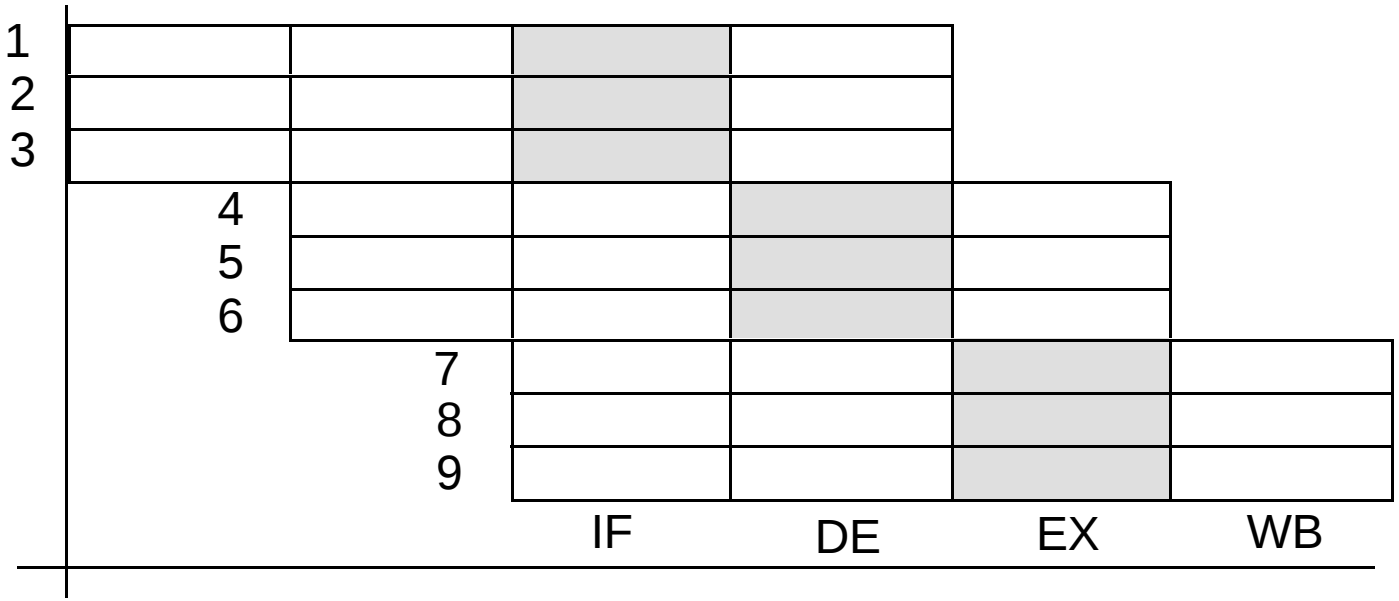
Beyond Scalar Limit

- Go beyond single instruction pipeline, achieve IPC > 1
- Dispatch multiple instructions per cycle
- Provide more generally applicable form of concurrency (not just vectors)
- Geared for sequential code that is hard to parallelize otherwise
- Exploit fine-grained or instruction-level parallelism (ILP)

Classifying ILP Machines

Super-scalar pipeline:

- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n instr / cycle (n x speedup?)

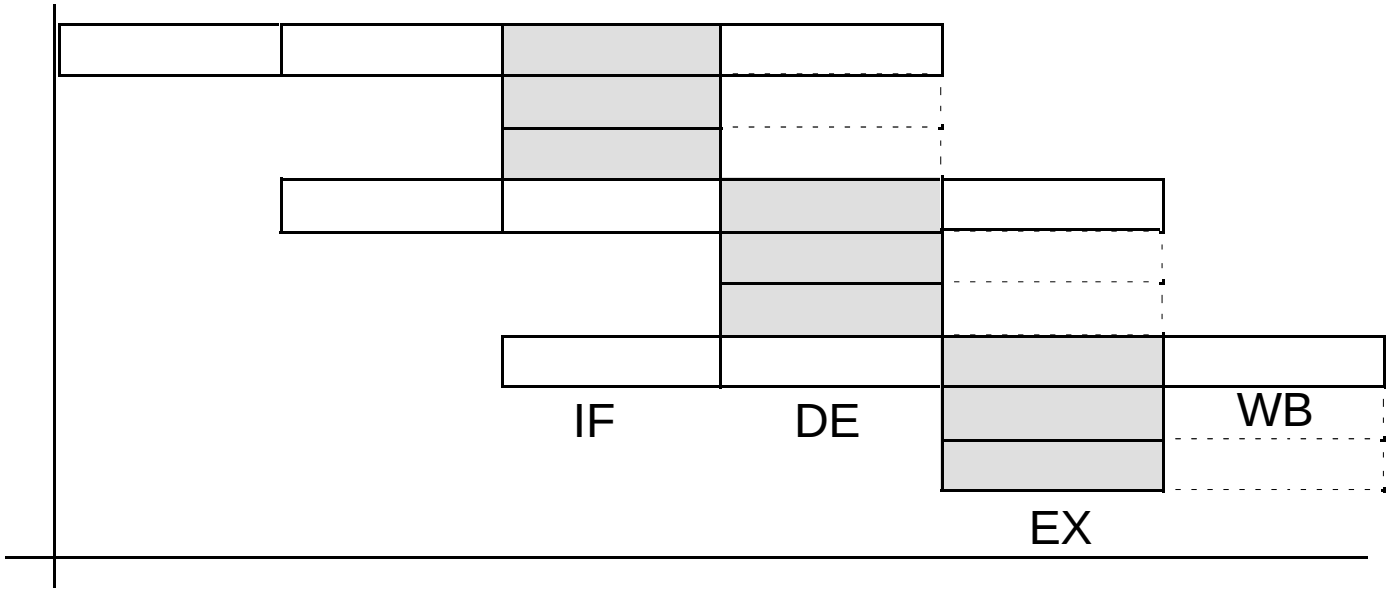


- Jouppi, WRL Reserch Report 89/7, 1989

Classifying ILP Machines

VLIW: Very Long Instruction Word:

- Issue parallelism = IP = n inst / cycle
- Operation latency = OP = 1 cycle
- Peak IPC = n instr / cycle = 1 VLIW / cycle



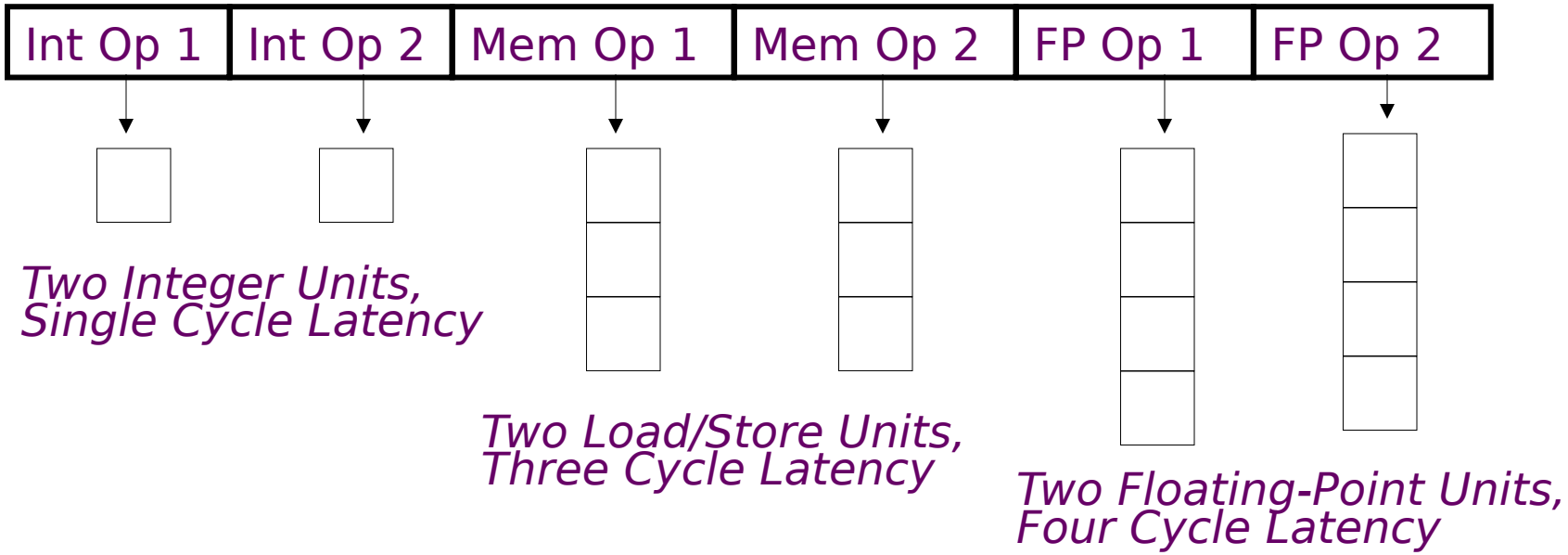
- Jouppi, WRL Reserch Report 89/7, 1989

Very Long Instruction Word Processor

VLIW: Idea and Motivation

- To overcome the difficulty of finding parallelism in machine-level object code.
- In a VLIW processor, multiple instructions are packed together and issued in parallel to an equal number of execution units.
- The compiler (not the processor) checks that there are only independent instructions executed in parallel.

VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no x-operation RAW check
  - No data use before data ready => no data interlocks

VLIW Compiler Responsibilities

The compiler:

- Schedules to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

Early VLIW Machines

- FPS AP120B (1976)
  - scientific attached array processor
  - first commercial wide instruction machine
  - hand-coded vector math libraries using software pipelining and loop unrolling
- Multiflow Trace (1987)
  - commercialization of ideas from Fisher's Yale group including "trace scheduling"
  - available in configurations with 7, 14, or 28 operations/instruction
  - 28 operations packed into a 1024-bit instruction word
- Cydrome Cydra-5 (1987)
  - 7 operations encoded in 256-bit instruction word
  - rotating register file

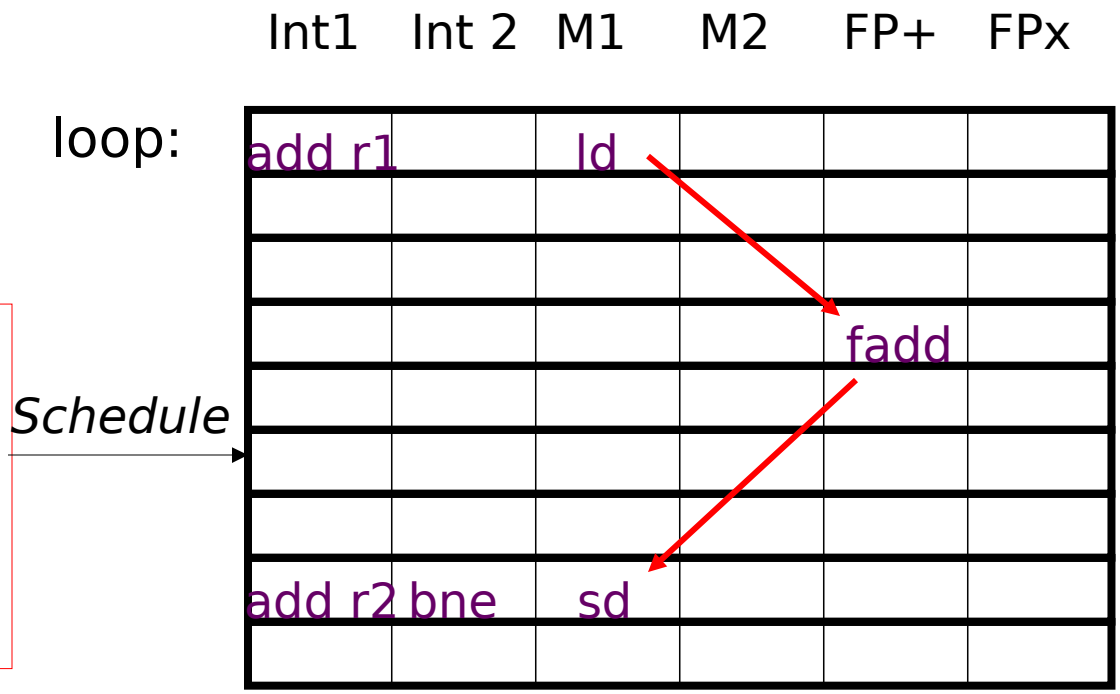
Loop Execution

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Compile

```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

Schedule



How many FP ops/cycle?

IPC = 1 fadd / 8 cycles = 0.125

Loop Unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop

Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule

	Int1	Int 2	M1	M2	FP+	FPx
loop:			ld f1			
			ld f2			
			ld f3			
	add r1		ld f4		fadd f5	
					fadd f6	
					fadd f7	
					fadd f8	
			sd f5			
			sd f6			
			sd f7			
	add r2	bne	sd f8			

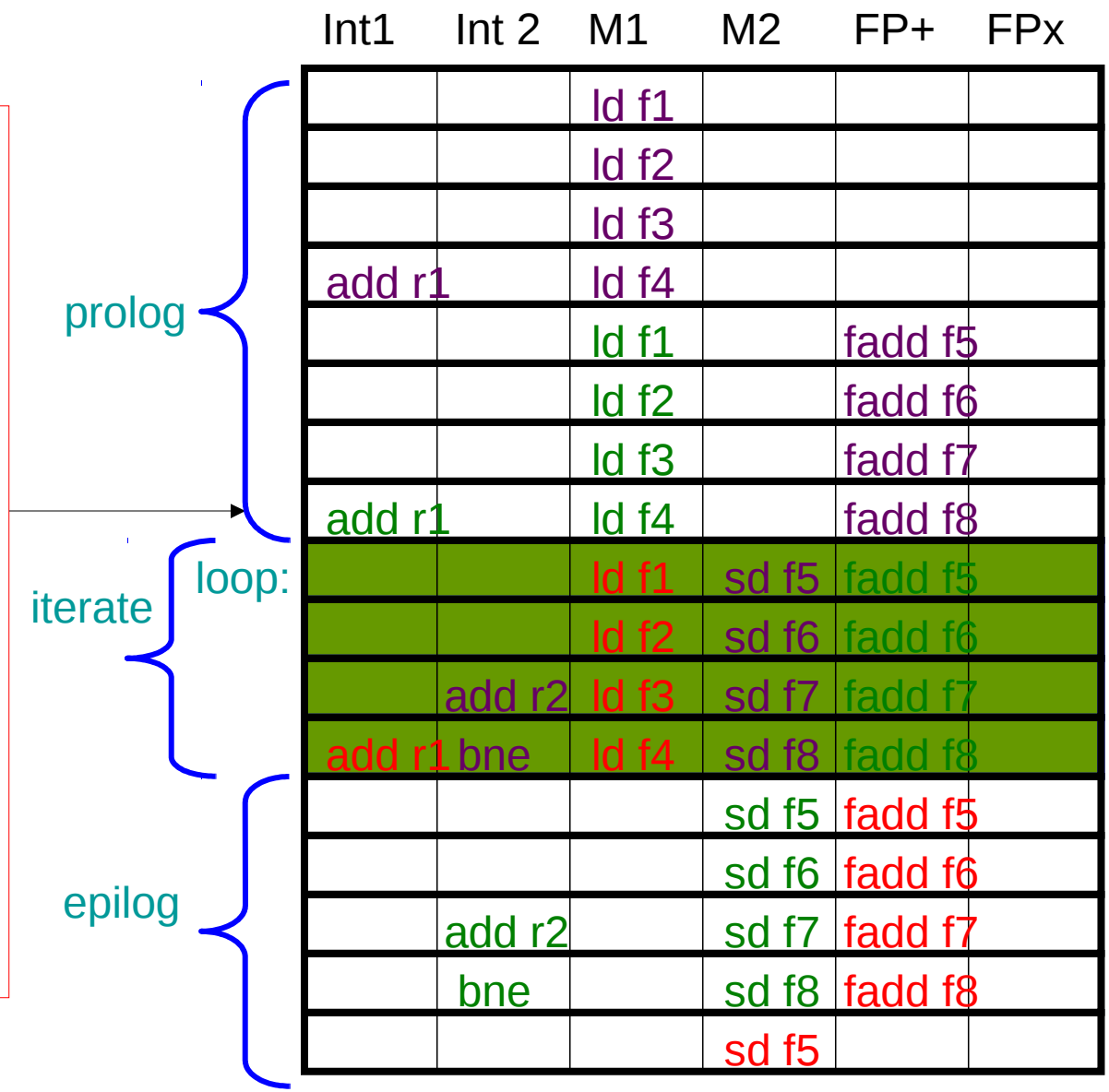
How many FLOPS/cycle?

IPC= 4 fadds / 11 cycles = 0.36

Software Pipelining

Unroll 4 ways first

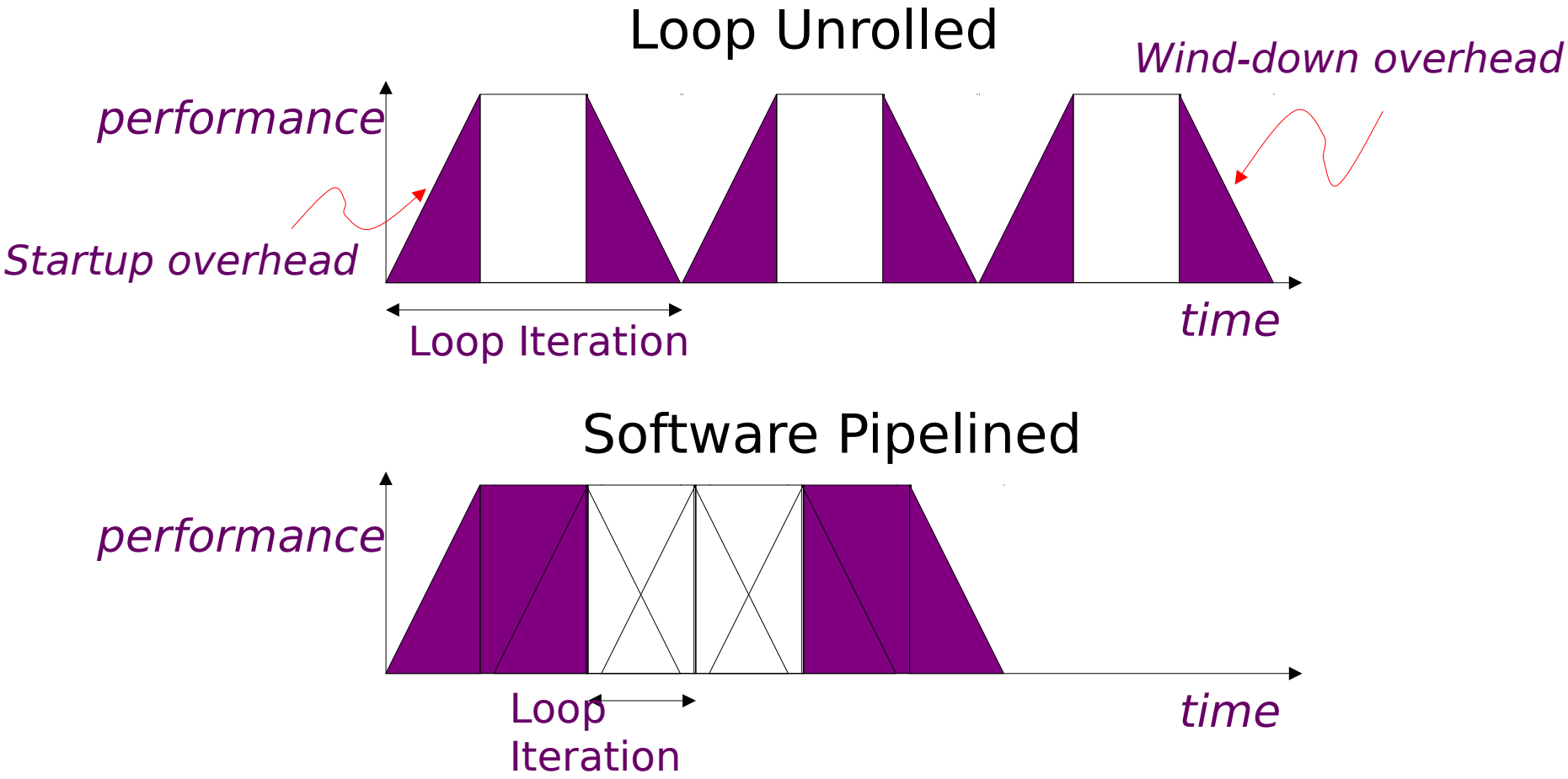
```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```



How many FLOPS/cycle?

4 fadds / 4 cycles = 1

Software Pipelining vs. Loop Unrolling



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

Reference:

- Jouppi, WRL Reserch Report 89/7, 1989

- Appendix C: Computer Architecture Quantitative Approach, 5th Edition. (Pipeline: Basic and Intermediate Concepts)

Next Lecture

Pipeline to continue...