Introduction To Pipelining

Moore's Law

Moore's Law says that the number of processors on a chip doubles about every 18 months.

Given the data on the following two slides, is this true?

Table 2-2. Key Features of Previous Generations of IA-32 Processors

Intel Processor	Date Intro-duced	Max. Clock Frequency at Intro-duction	Transis-tors per Die	Register Sizes[1]	Ext. Data Bus Size[2]	Max. Extern. Addr. Space	Caches
8086	1978	8 MHz	29 K	16 GP	16	1 MB	None
Intel 286	1982	12.5 MHz	134 K	16 GP	16	16 MB	Note 3
Intel386 DX Processor	1985	20 MHz	275 K	32 GP	32	4 GB	Note 3
Intel486 DX Processor	1989	25 MHz	1.2 M	32 GP 80 FPU	32	4 GB	L1: 8KB
Pentium Processor	1993	60 MHz	3.1 M	32 GP 80 FPU	64	4 GB	L1:16KB
Pentium Pro Processor	1995	200 MHz	5.5 M	32 GP 80 FPU	64	64 GB	L1: 16KB L2: 256KB or 512KB
Pentium II Processor	1997	266 MHz	7 M	32 GP 80 FPU 64 MMX	64	64 GB	L1: 32KB L2: 256KB or 512KB
Pentium III Processor	1999	500 MHz	8.2 M	32 GP 80 FPU 64 MMX 128 XMM	64	64 GB	L1: 32KB L2: 512KB

NOTES:

1. The register size and external data bus size are given in bits. Note also that each 32-bit general-purpose (GP) registers can be addressed as an 8- or a 16-bit data registers in all of the processors
2. Internal data paths that are 2 to 4 times wider than the external data bus for each processor.

Table 2-1. Key Features of Most Recent IA-32 Processors

Intel Processor	Date Intro-duced	Micro-Architecture	Clock Frequency at Intro-duction	Transis-tors Per Die	Register Sizes[1]	System Bus Band-width	Max. Extern. Addr. Space	On-Die Caches[2]
Pentium III and Pentium III Xeon Processors[3]	1999	P6	700 MHz	28 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	Up to 1.06 GB/s	64 GB	32-KB L1; 256-KB L2
Pentium 4 Processor	2000	Intel NetBurst Micro-architecture	1.50 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K µop Execution Trace Cache; 8KB L1; 256-KB L2
Intel Xeon Processor	2001	Intel NetBurst Micro-architecture	1.70 GHz	42 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K µop Trace Cache; 8-KB L1; 256-KB L2
Intel Xeon Processor[4]	2002	Intel NetBurst Micro-architecture; Hyper-Threading Technology	2.20 GHz	55 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K µop Trace Cache; 8-KB L1; 512-KB L2
Intel® Xeon™ Processor MP[4]	2002	Intel NetBurst Micro-architecture; Hyper-Threading Technology	1.60 GHz	108 M	GP: 32 FPU: 80 MMX: 64 XMM: 128	3.2 GB/s	64 GB	12K µop Trace Cache; 8-KB L1; 256-KB L2; 1-MB L3

NOTES

1. The register size and external data bus size are given in bits.
2. First level cache is denoted using the abbreviation L1, 2nd level cache is denoted as L2
3. Intel Pentium III and Pentium III Xeon processors, with advanced transfer cache and built on 0.18 micron process technology, were introduced in October 1999.
4. Hyper-Threading technology is implemented with two logical processors.

Intel Architecture

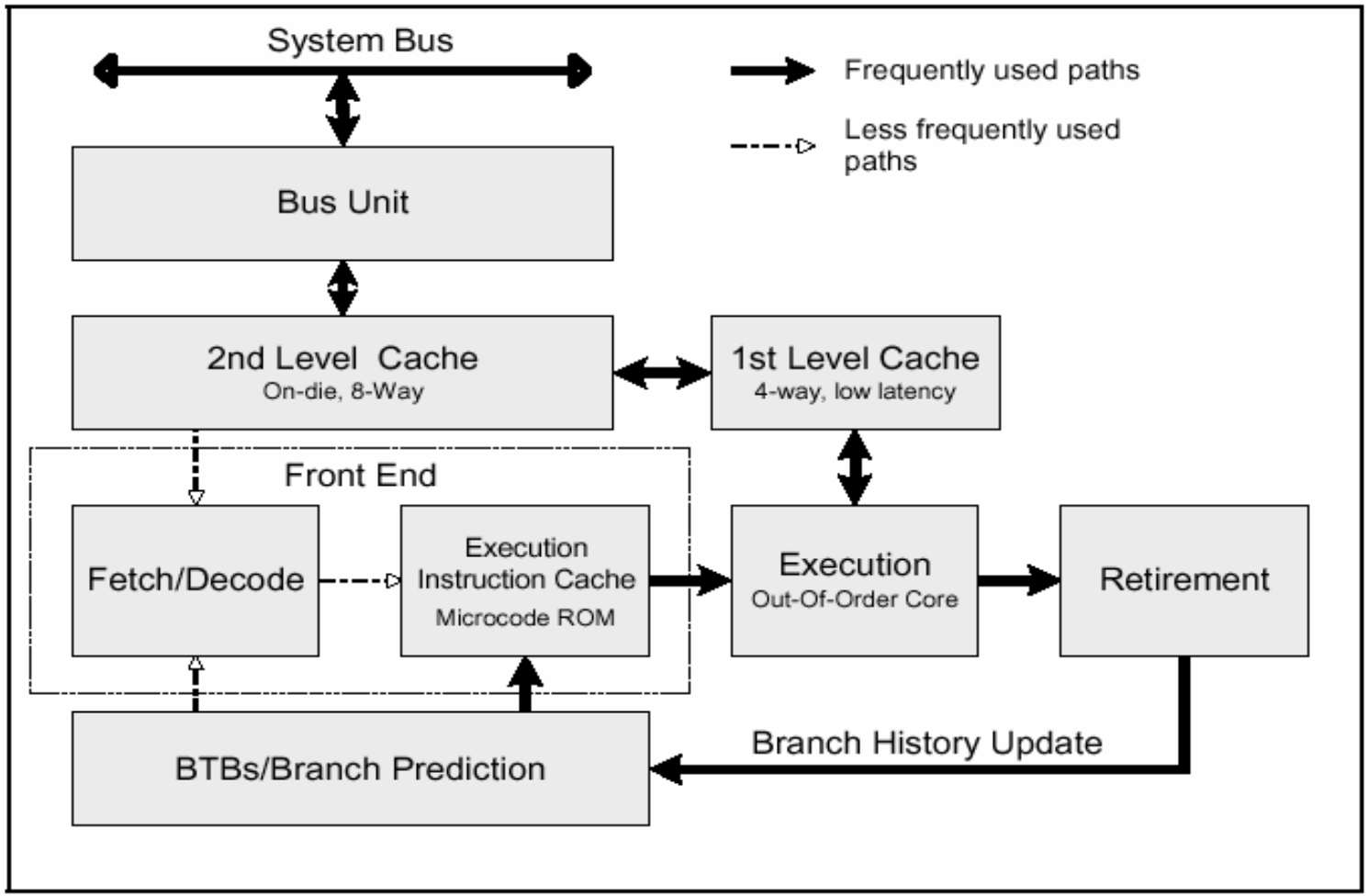


Figure 2-1. The P6 Processor Micro-Architecture with Advanced Transfer Cache Enhancement

Intel Architecture

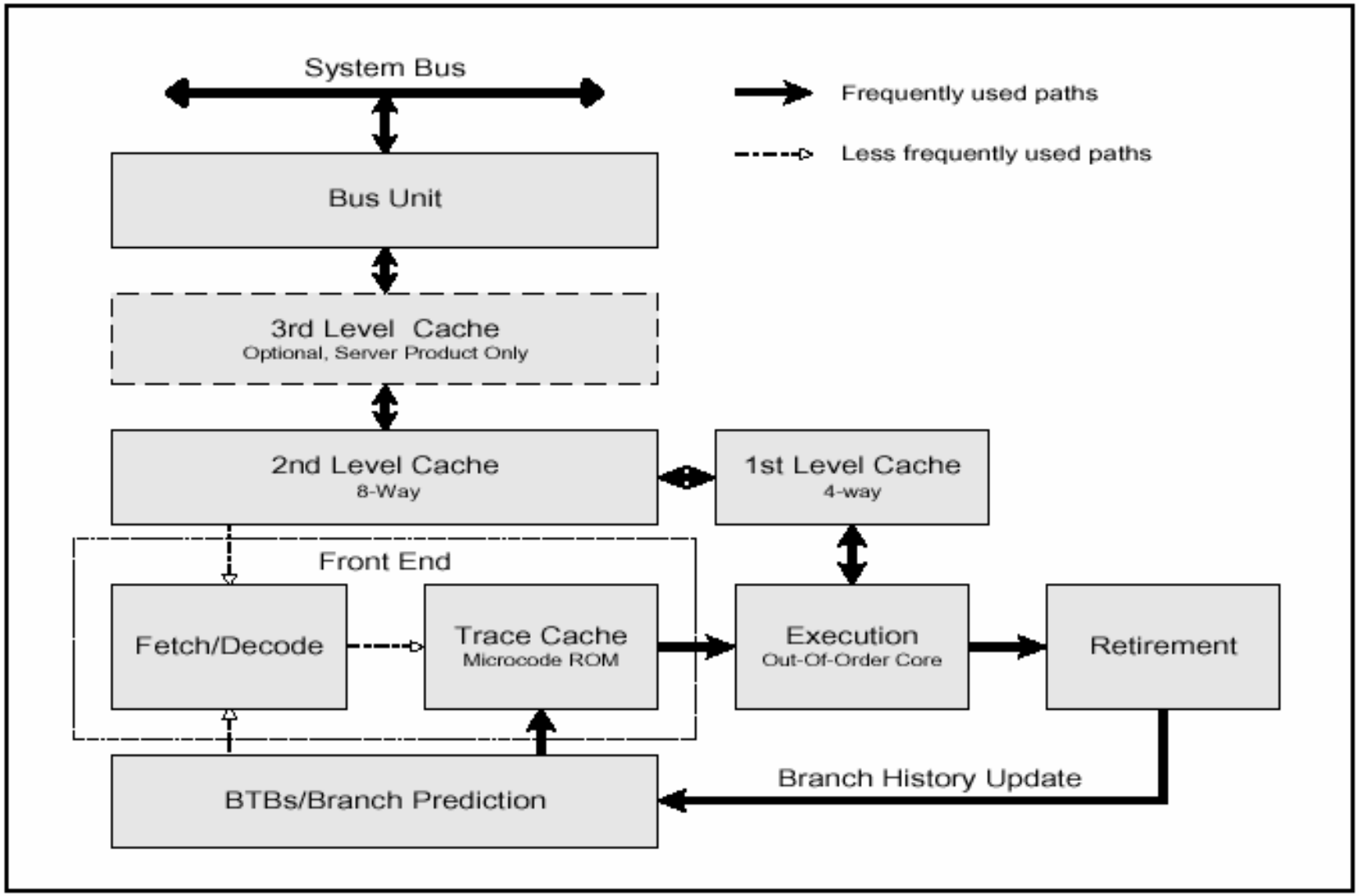


Figure 2-2. The Intel NetBurst Micro-Architecture

Intel Architecture

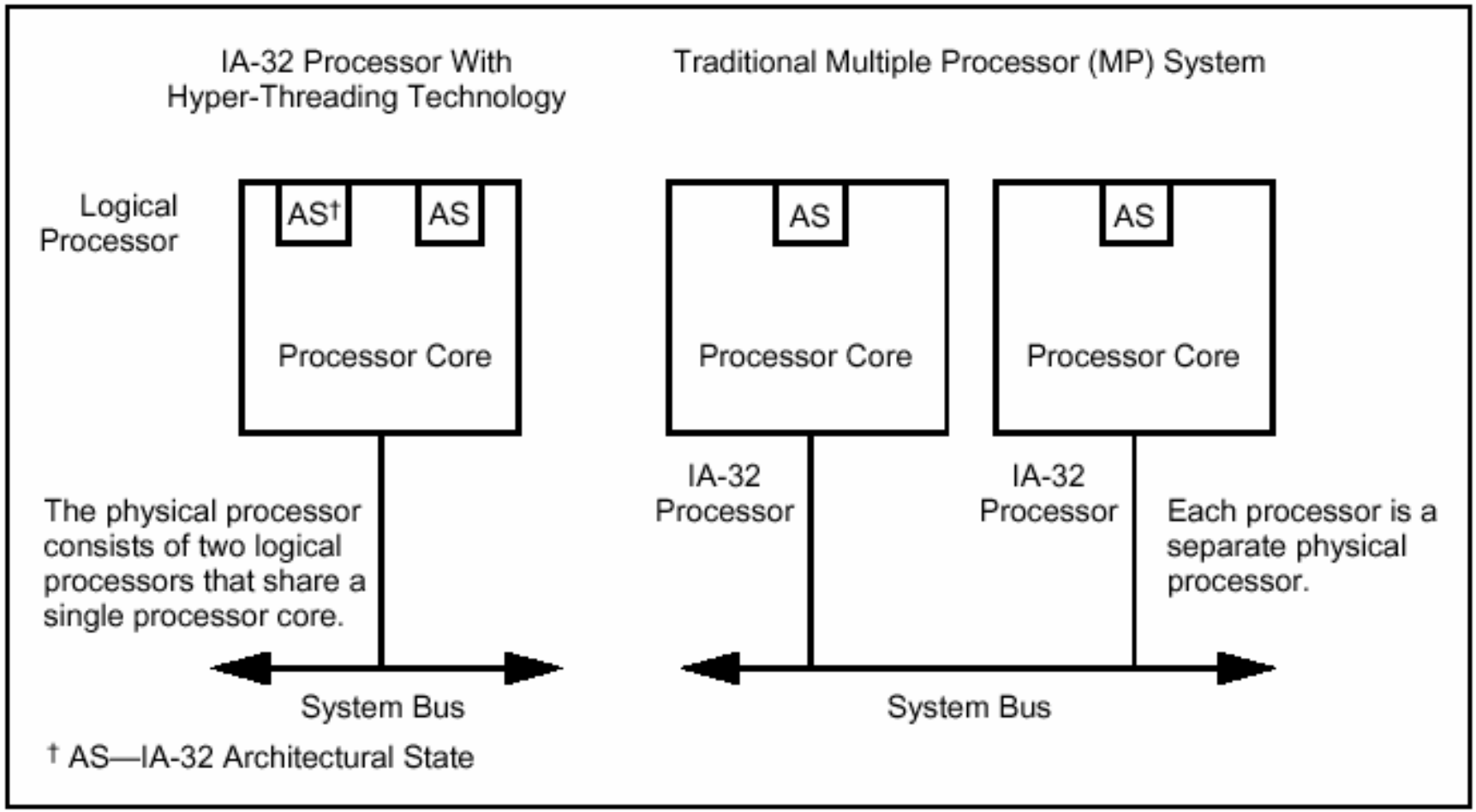
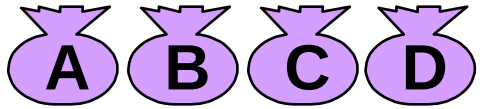
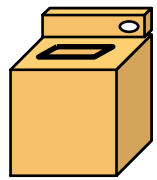


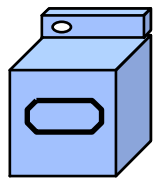
Figure 2-3. Comparison of an IA-32 Processor with Hyper-Threading Technology and a Traditional Dual Processor System.

Pipelining is Natural!

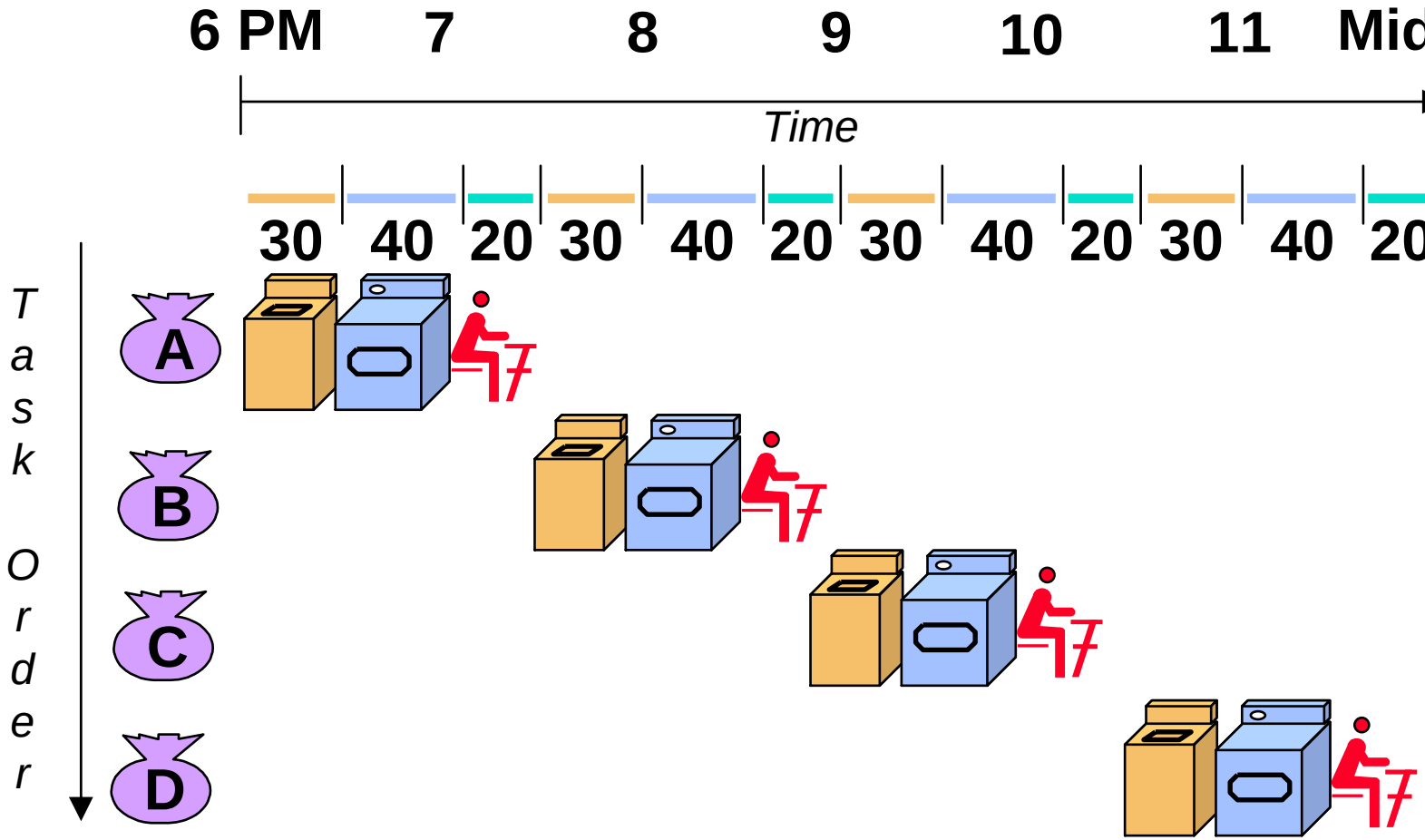


- **Laundry Example**
- **Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold**
- **Washer takes 30 minutes**
- **Dryer takes 40 minutes**
- **“Folder” takes 20 minutes**



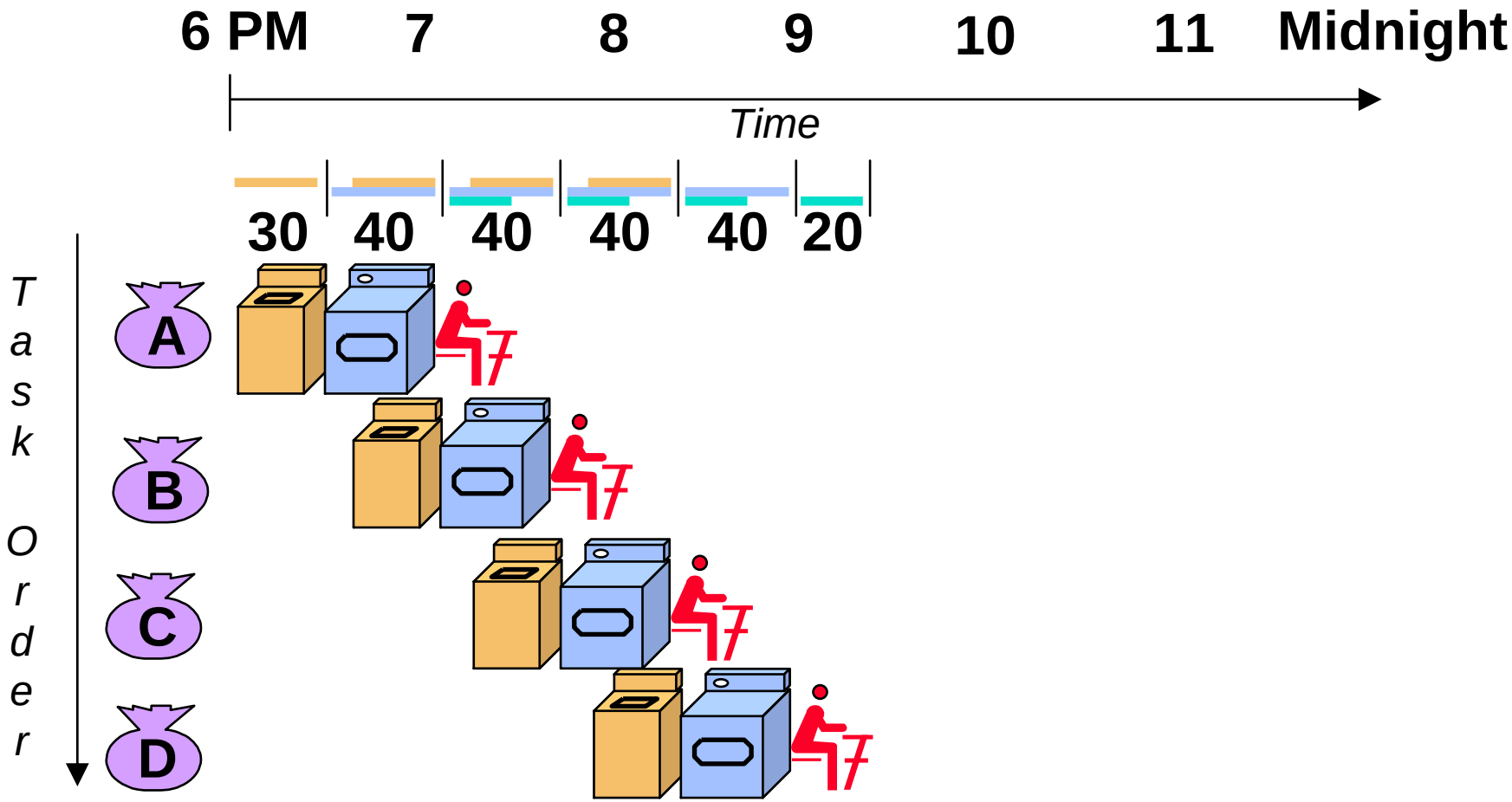


Sequential Laundry



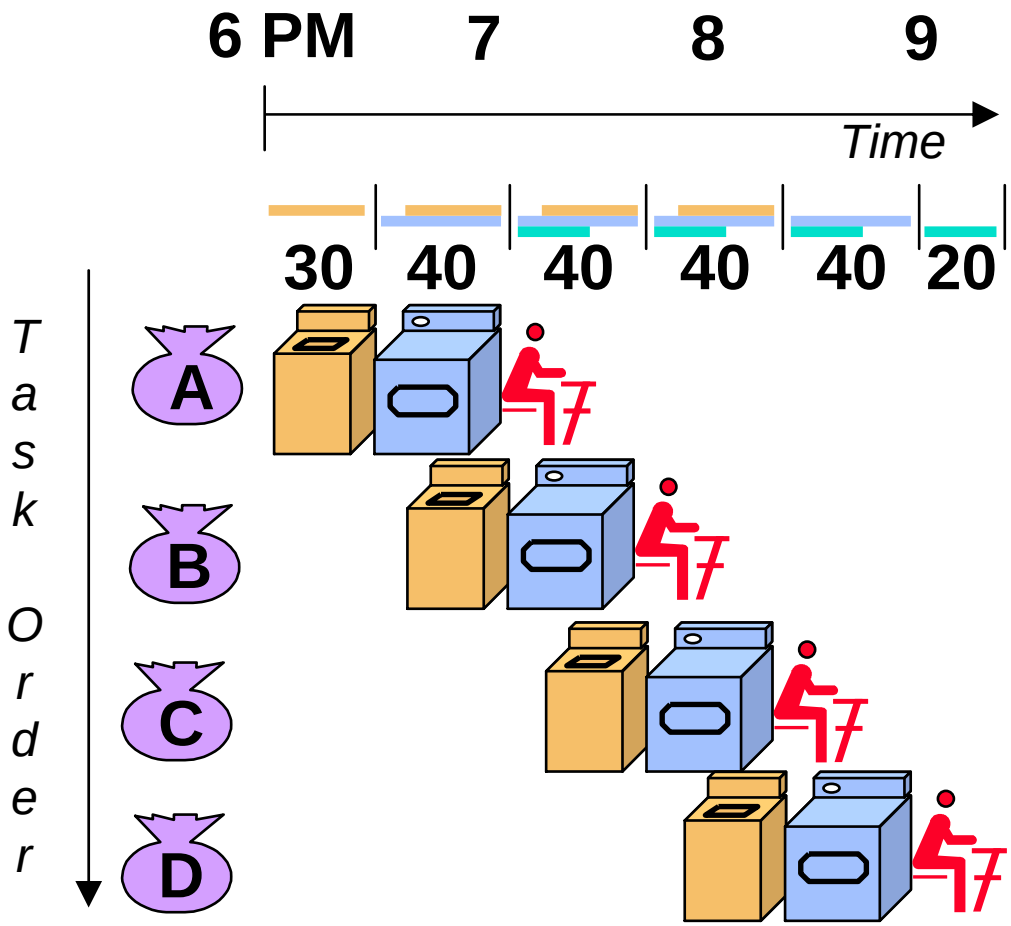
- **Sequential laundry takes 6 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

Pipelined Laundry: Start work ASAP



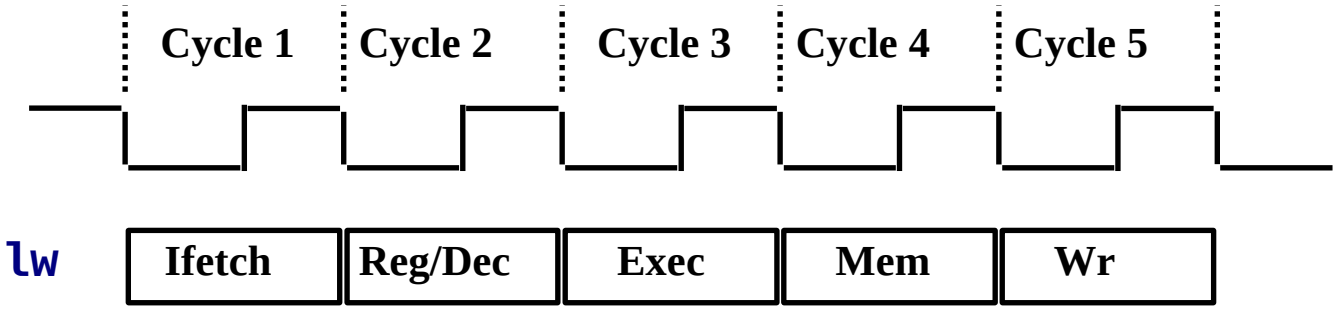
- **Pipelined laundry takes 3.5 hours for 4 loads**

Pipelining Lessons



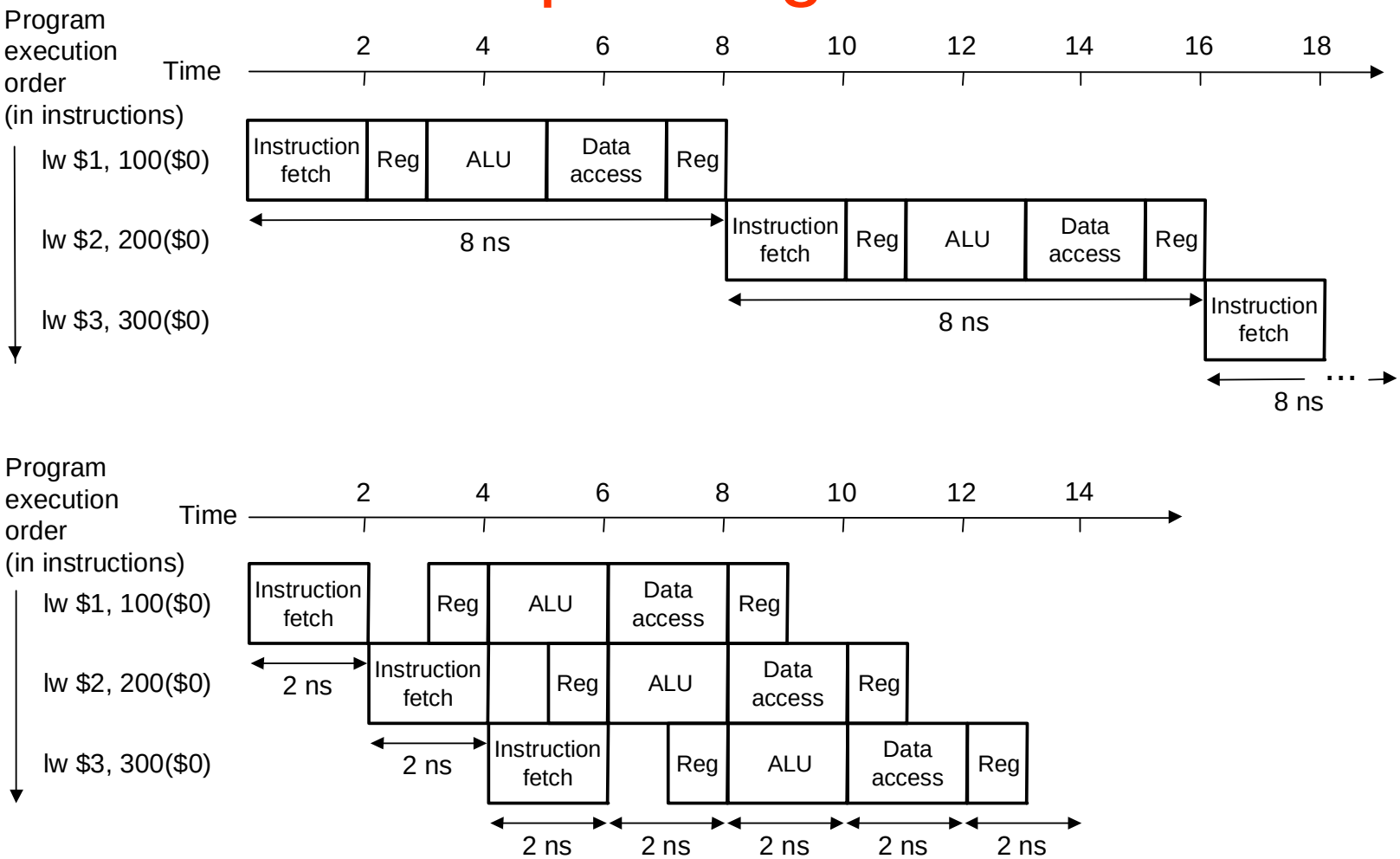
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

The Five Stages of An Instruction



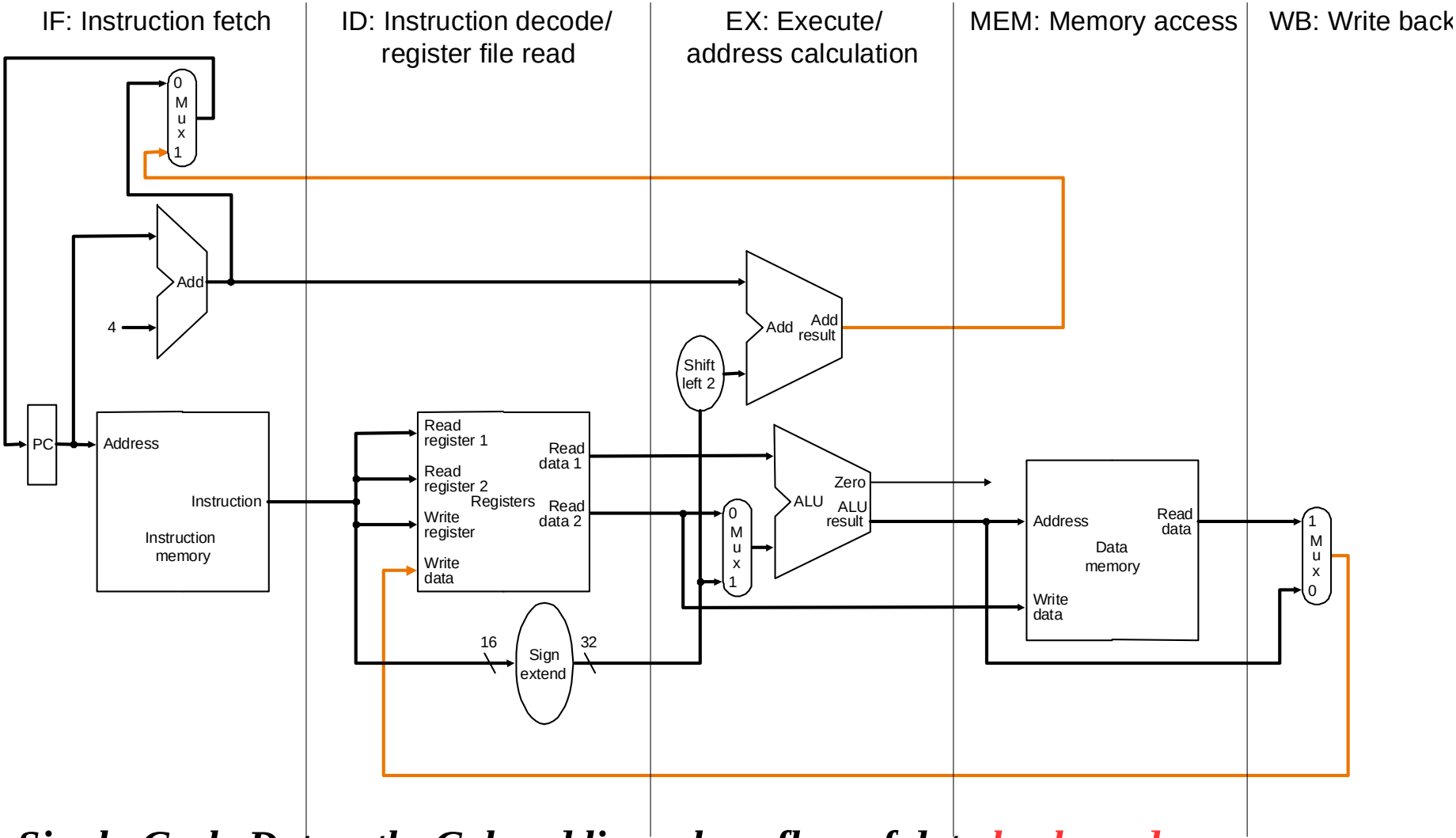
- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Read the data from the Data Memory**
- **Wr: Write the data back to the register file**

Pipelining



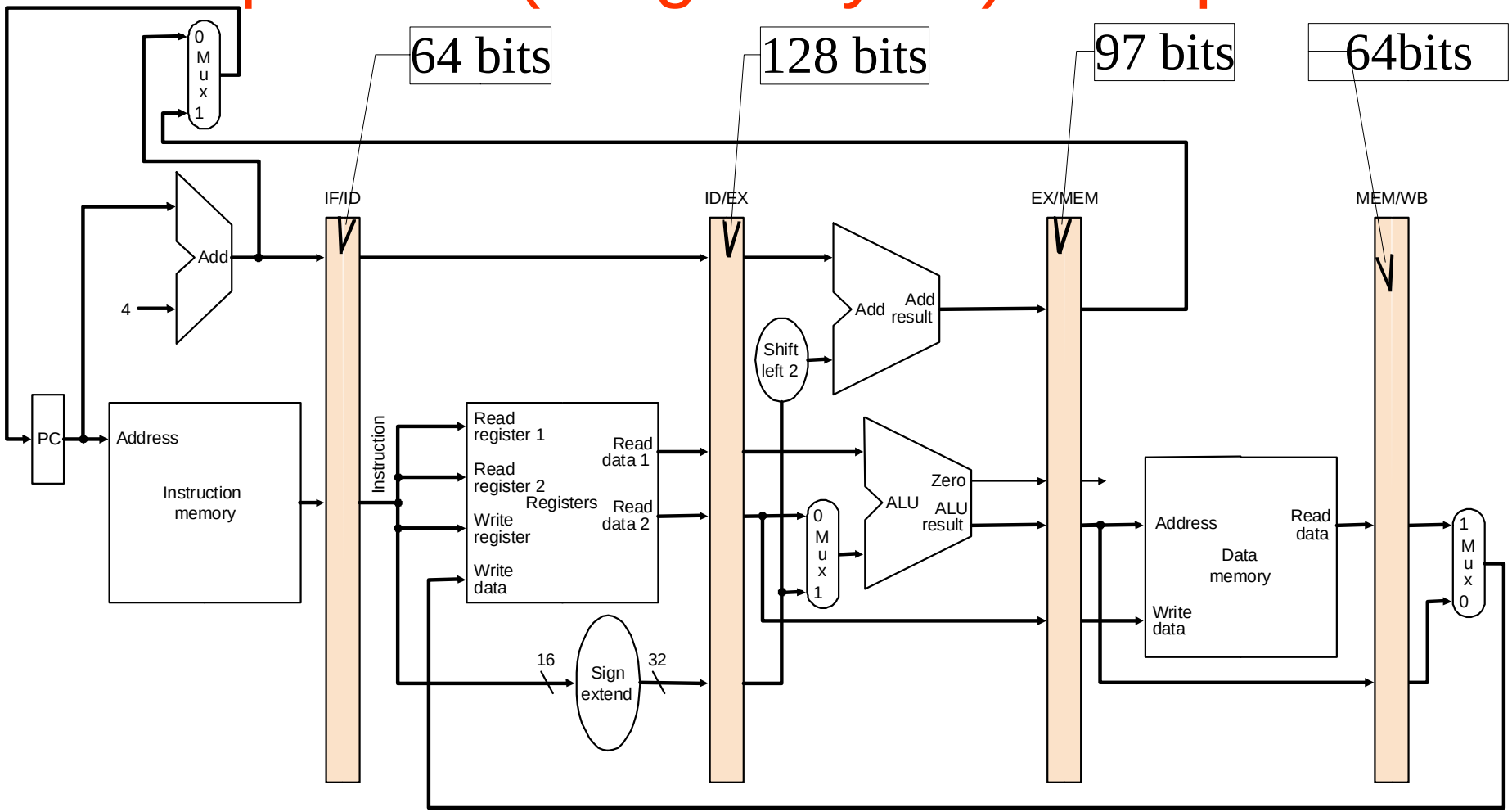
- **Improve performance by increasing throughput**
- ***Ideal speedup is number of stages in the pipeline. Do we achieve this?***

Basic Idea



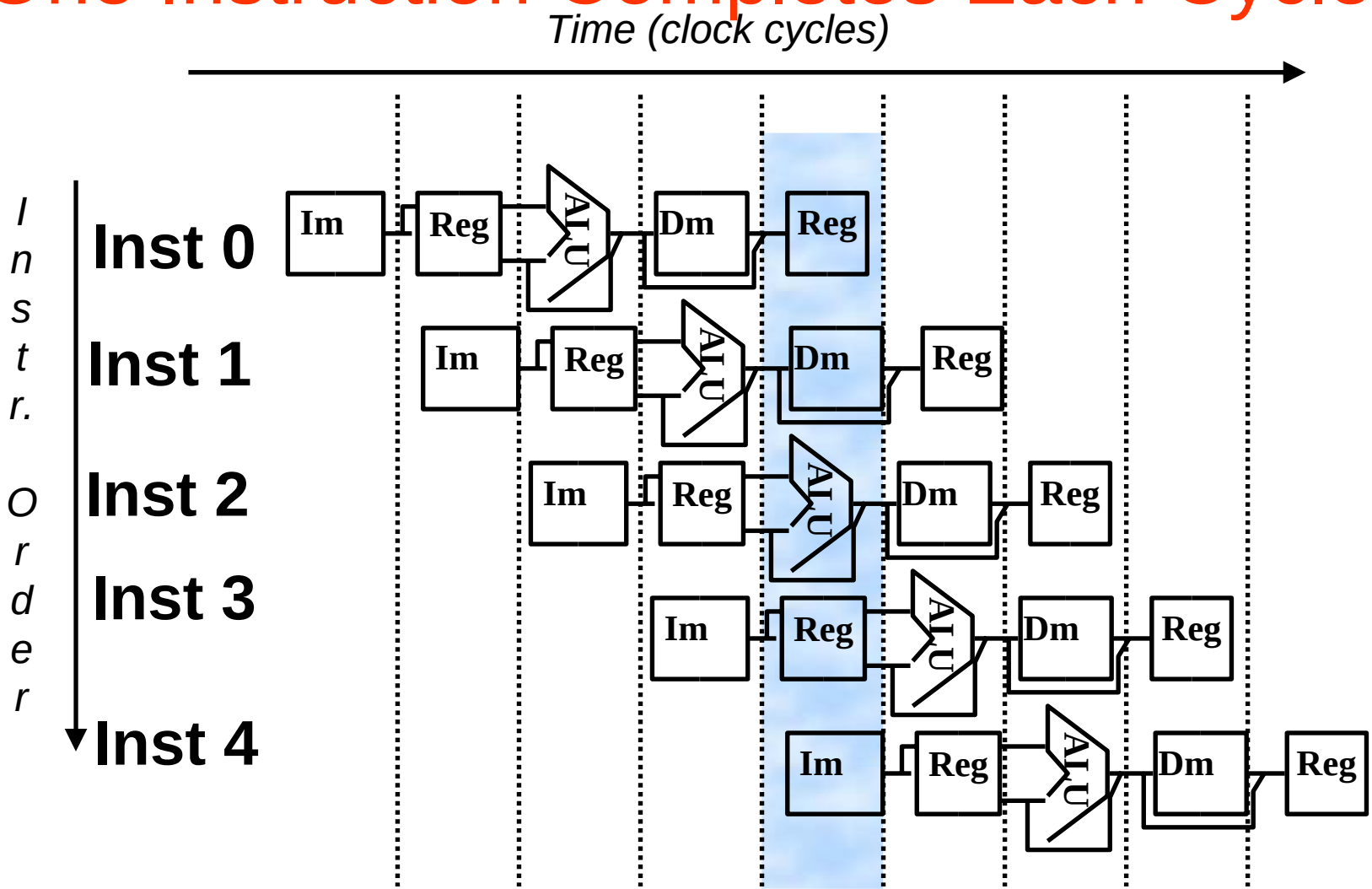
- ***Single-Cycle Datapath; Colored lines show flow of data backwards.***
- ***What do we need to add to split the datapath into stages?***

Pipelined (Single-Cycle) Datapath



- **Pipeline registers (colored), separate the datapath stages.**
- **Must be wide enough to store data, control and conditions as they flow downstream.**

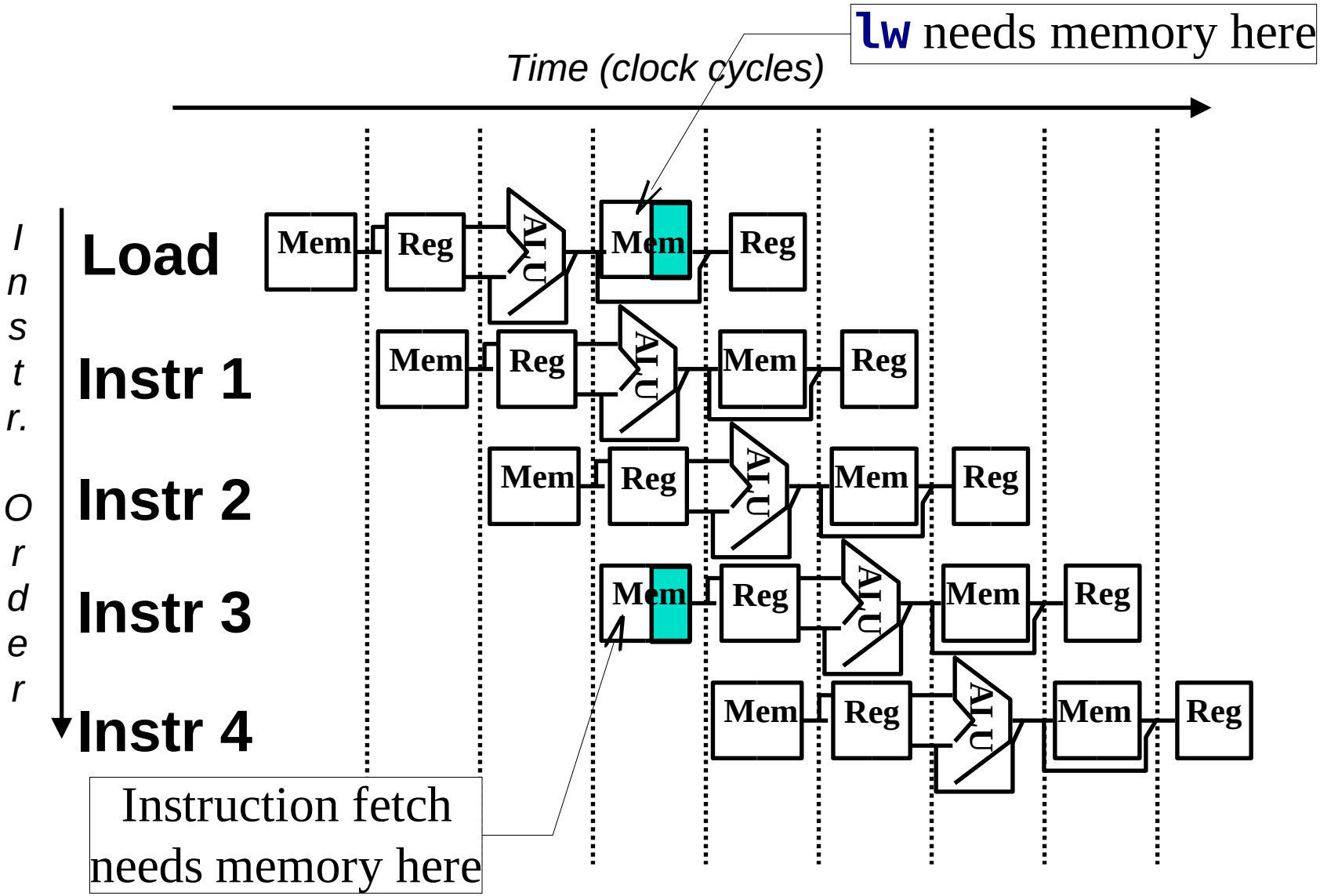
Why Pipeline?
One Instruction Completes Each Cycle!



Can pipelining get us into trouble?

- **Yes: Pipeline Hazards**
  - structural hazards: attempt to use the same resource two different ways at the same time
    - *e.g.*, combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
  - control hazards: attempt to make a decision before condition is evaluated
    - *e.g.*, washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
    - branch instructions
  - data hazards: attempt to use item before it is ready
    - *e.g.*, one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
    - instruction depends on result of prior instruction still in the pipeline
- **Can always resolve hazards by waiting**
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory Is a Structural Hazard

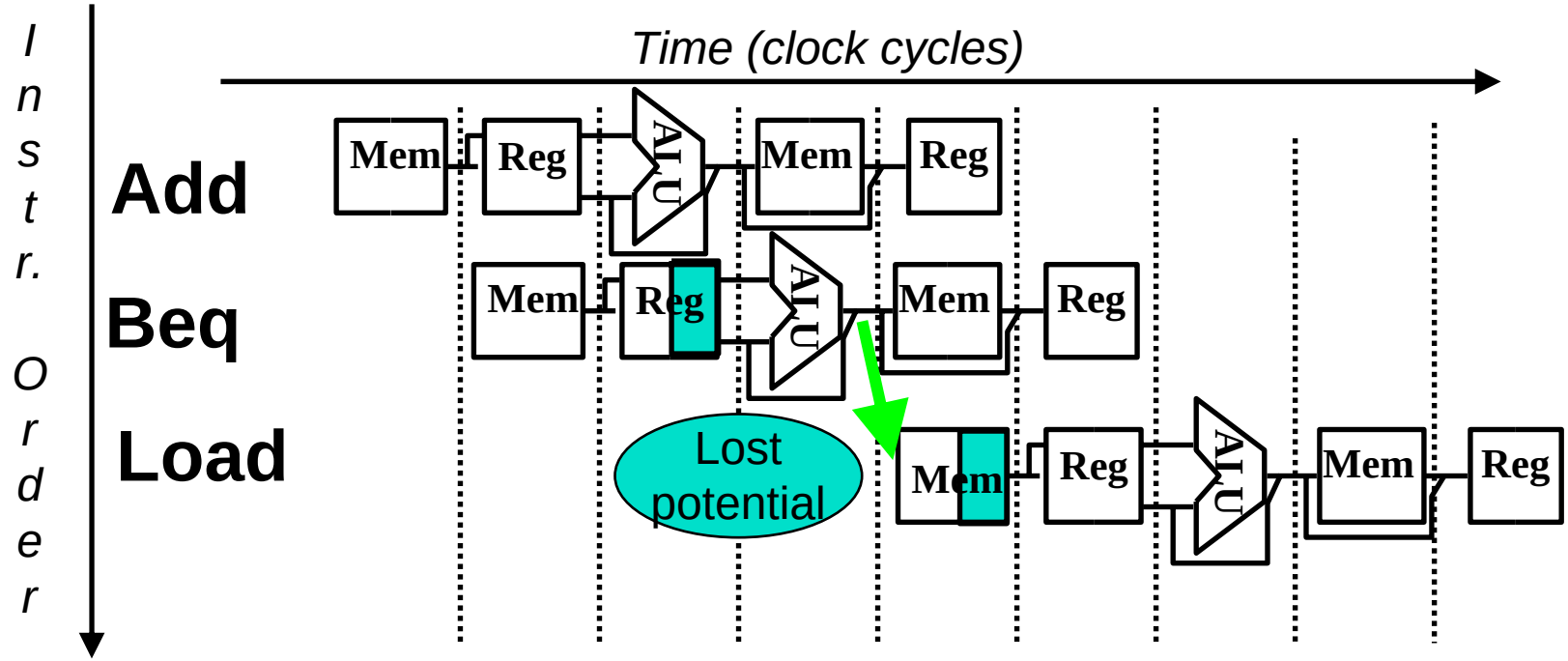


Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards Limit Performance

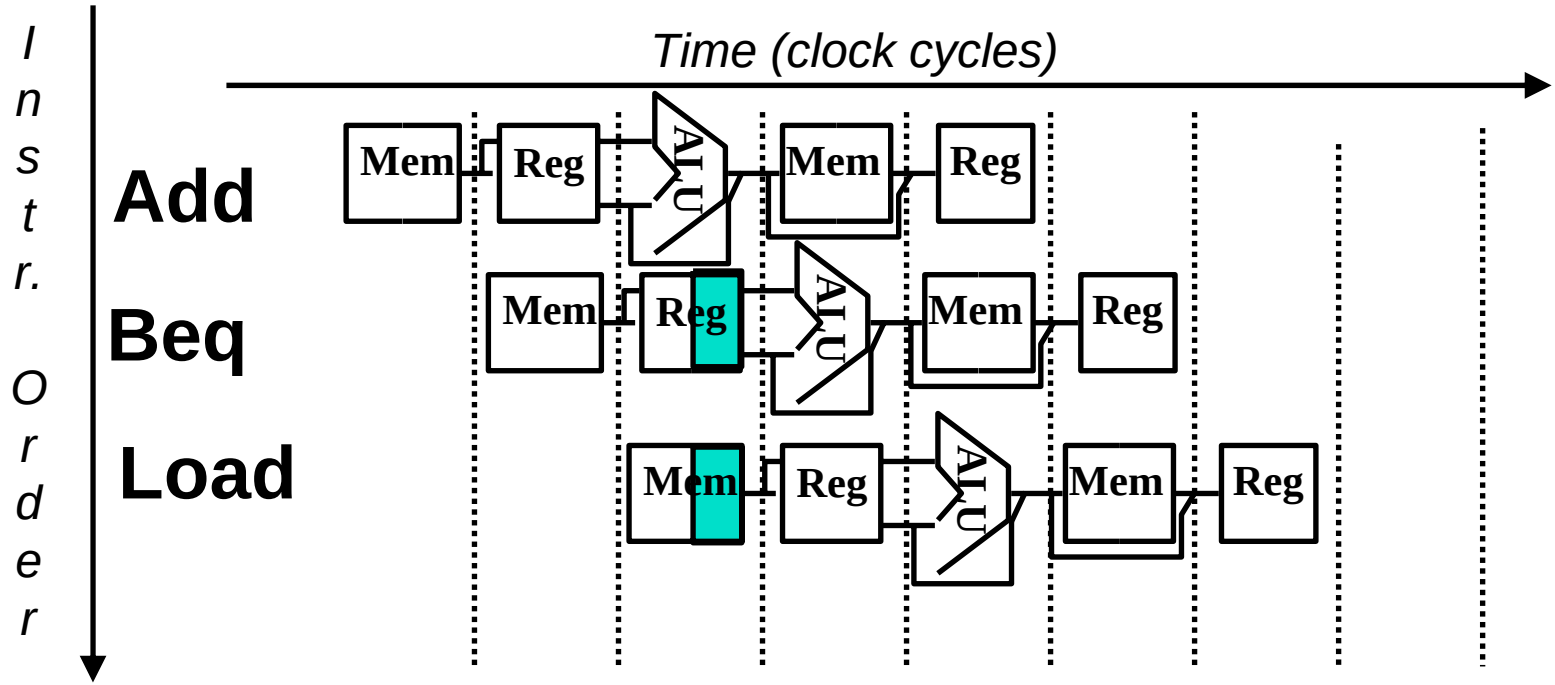
- **Example: if 1.3 memory accesses per instruction and only one memory access per cycle then.**
  - average CPI ≥ 1.3 ;
  - otherwise, resource is more than 100% utilized .

Control Hazard Solution #1: Stall



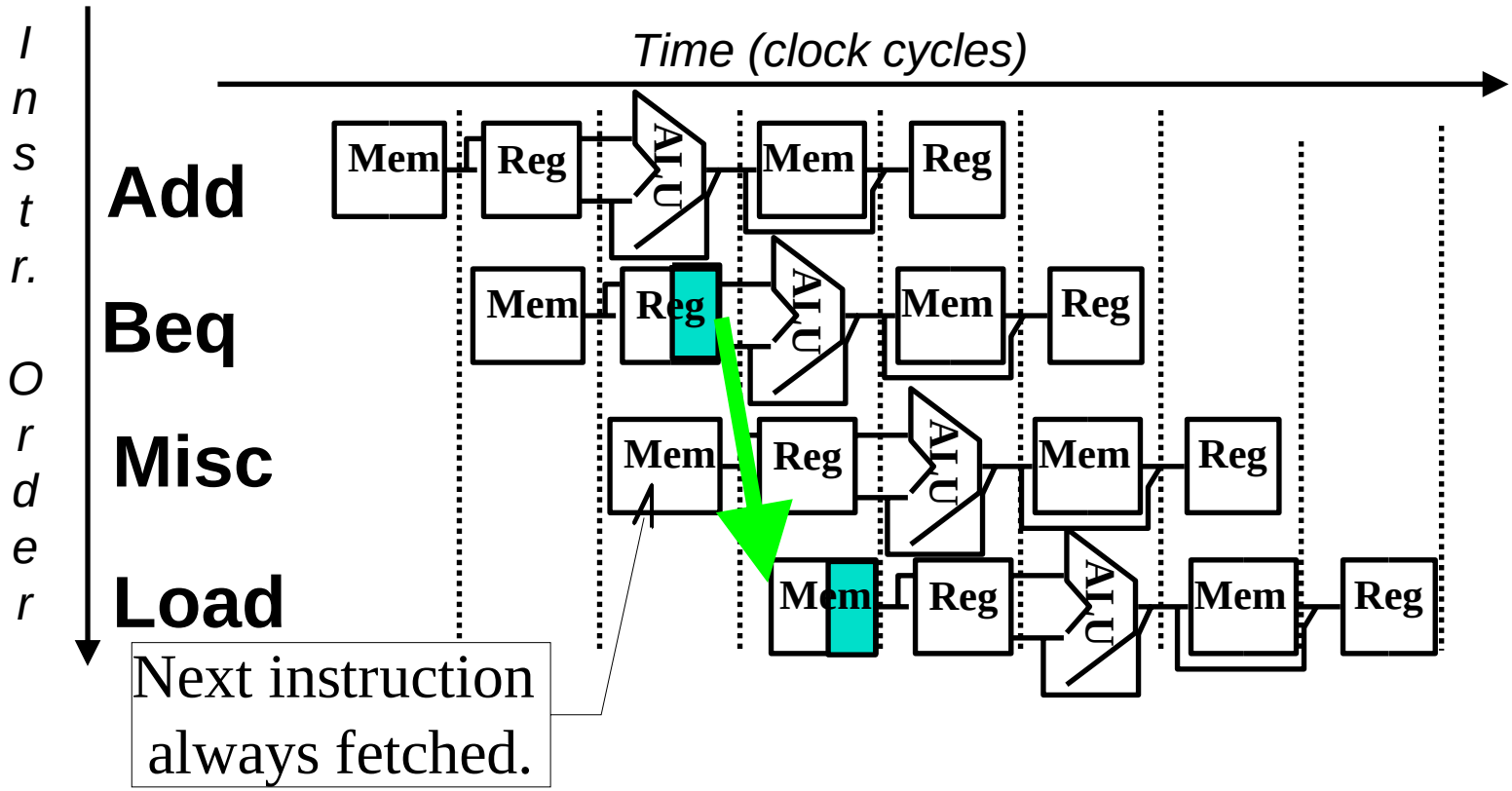
- **Stall: wait until decision is clear (conditional branching).**
- ***Impact*: 2 lost cycles (*i.e.*, 3 clock cycles per branch instruction) => slow.**
- **Move decision to end of decode.**
  - save 1 cycle per branch.

Control Hazard Solution #2: Predict



- **Predict: guess one direction then back up if wrong**
- **Impact: 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)**
  - Need to "Squash" and restart following instruction if wrong
  - Produce CPI on branch of (1 × .5 + 2 × .5) = 1.5
  - Total CPI might then be: 1.5 × .2 + 1 × .8 = 1.1 (20% branch)
- **More dynamic scheme: history of 1 branch (- 90%)**

Control Hazard Solution #3: Delayed Branch



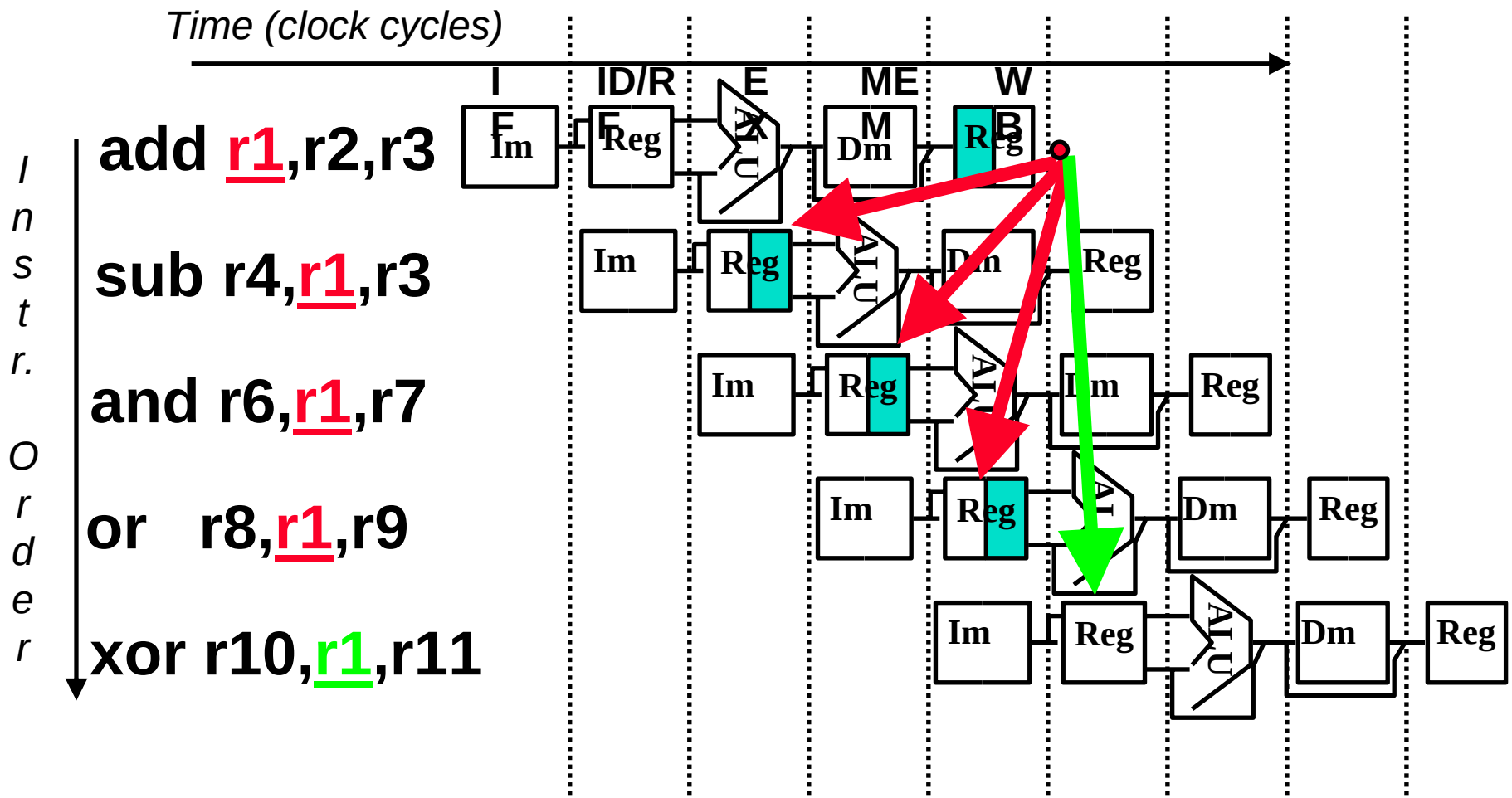
- **Delayed Branch: Redefine branch behavior (takes place *after* next instruction)**
- **Impact: 0 clock cycles per branch instruction if can find instruction to put in "slot" (- 50% of time)**
- **As launch more instruction per clock cycle, less useful**

Data Hazard on R1

```
add r1, r2, r3
sub r4, r1, r3
and r6, r1, r7
or  r8, r1, r9
xor r10, r1, r11
```

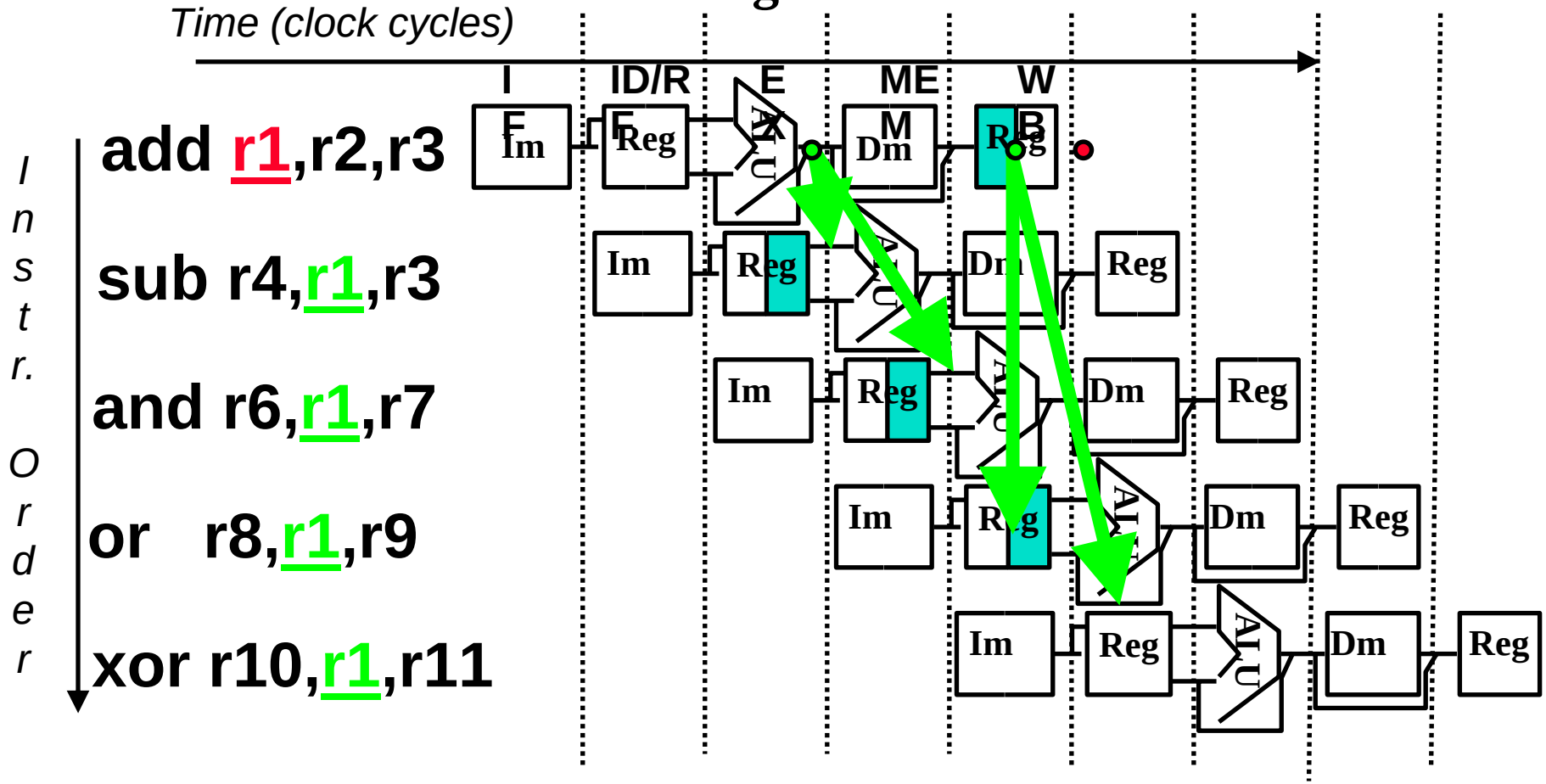
Data Hazard on R1:

- **Dependencies "backwards" in time are hazards**



Data Hazard Solution: Forwarding (or Bypassing)

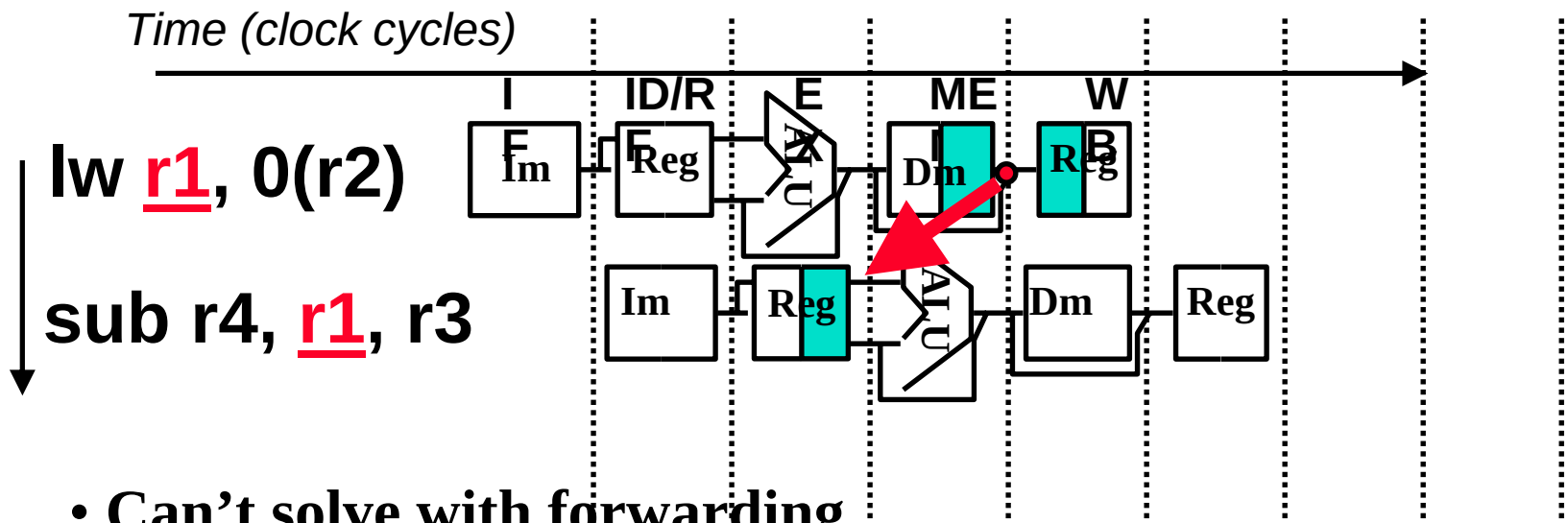
- **"Forward" result from one stage to another**



- **"or" OK if define read/write of register-file properly**

Forwarding (or Bypassing): What About Loads?

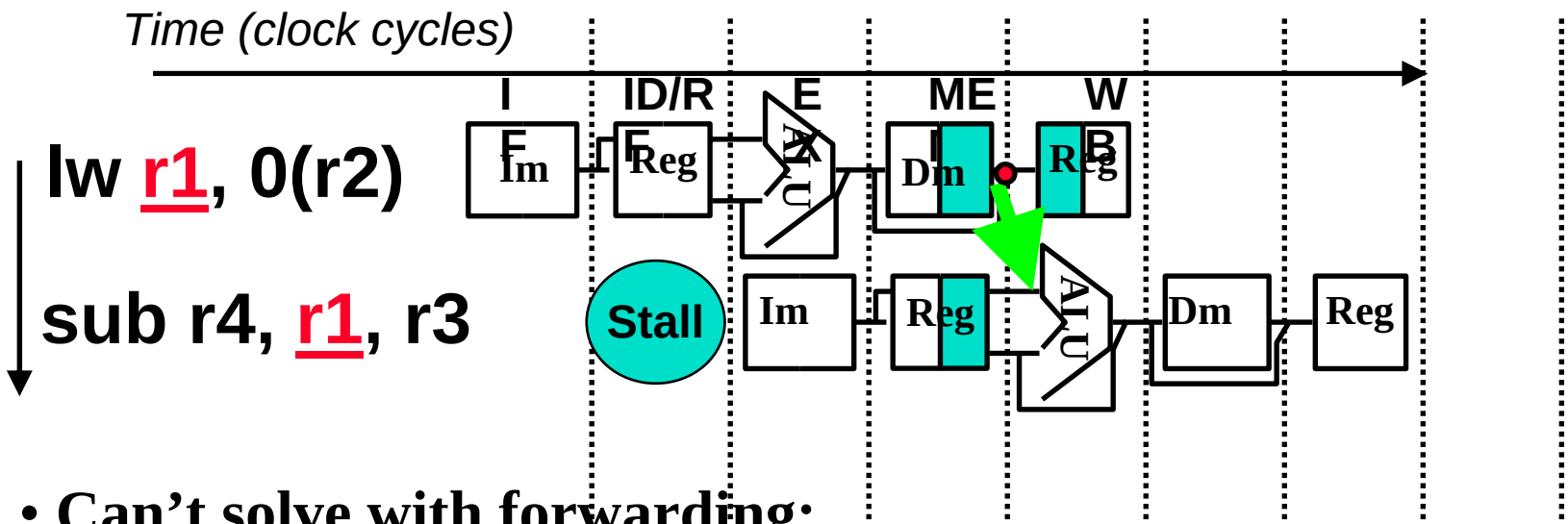
- **Dependencies "backwards" in time are hazards**



- **Can't solve with forwarding.**
- **Must delay/stall instruction dependent on loads**

Forwarding (or Bypassing What About Loads

- **Dependencies backwards in time are hazards**



- **Can't solve with forwarding:**
- **Must delay/stall instruction which dependent on loads**
- **Then directly forward data to resource requesting it (ALU)**

Summary: Pipelining

- **Reduce CPI by overlapping many instructions.**
  - Average throughput of approximately 1 CPI with fast clock.

- **What makes it easy:**
  - all instructions are the same length;
  - just a few instruction formats;
  - memory operands appear only in loads and stores.

  .
- **What makes it hard?**
  - structural hazards: suppose we had only one memory;
  - control hazards: need to worry about branch instructions;
  - data hazards: an instruction depends on a previous instruction.

  .

Summary

- **Microprogramming is a fundamental concept**
  - implement an instruction set by building a very simple processor and interpreting the instructions
  - essential for very complex instructions and when few register transfers are possible
  - *Control design reduces to Microprogramming*