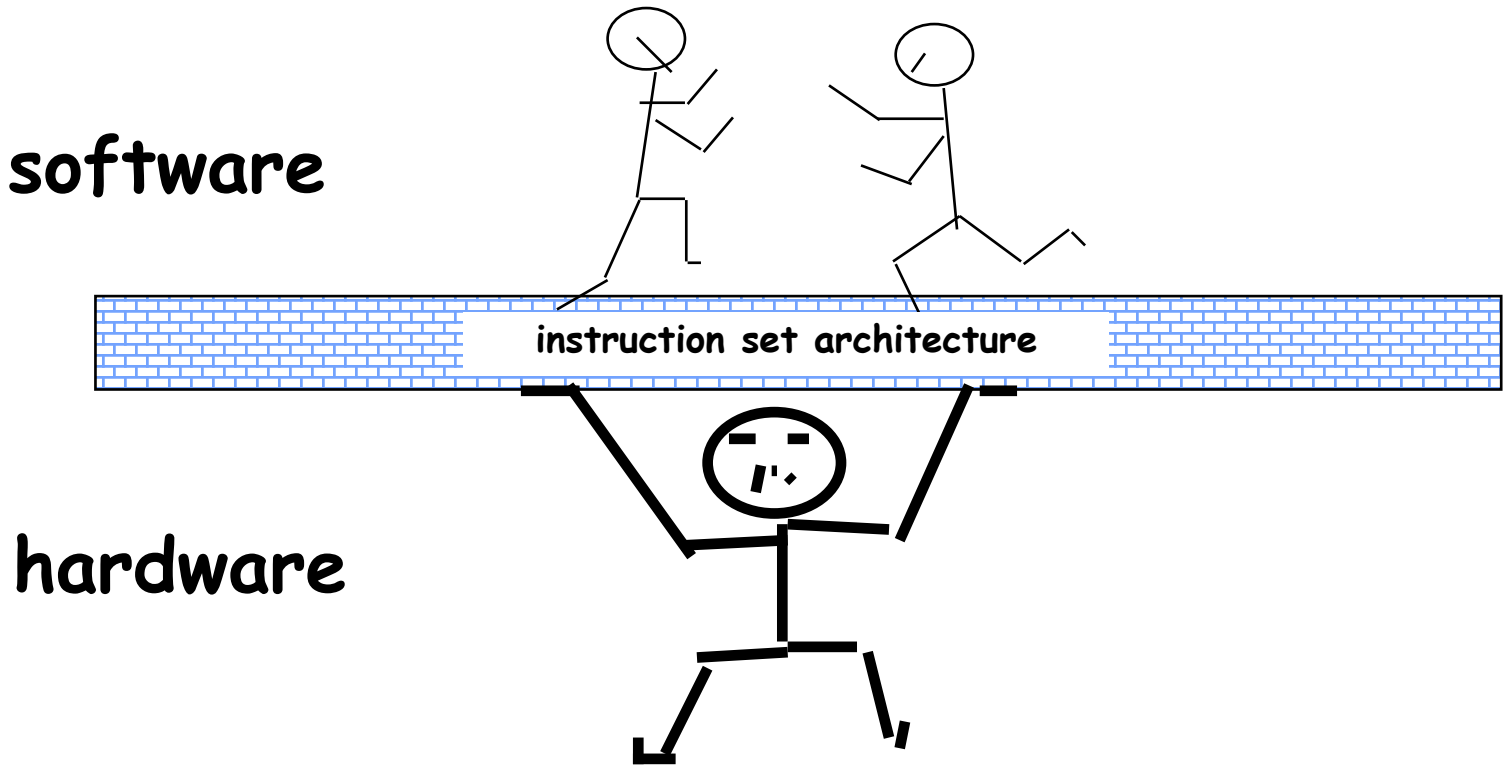
ENGG 5101
Advanced Computer Architecture

Lecture 03 – Basic Concepts

XU, Qiang (Johnny) 徐強

The Instruction Set Architecture (ISA)



The interface description separating the software and hardware

ISA Design Considerations

- Functionality and flexibility for software development
- Implementation efficiency in available technology
- Backward compatibility

How to design a long-lasting ISA under changes in usage and technology?

Fundamental Instructions

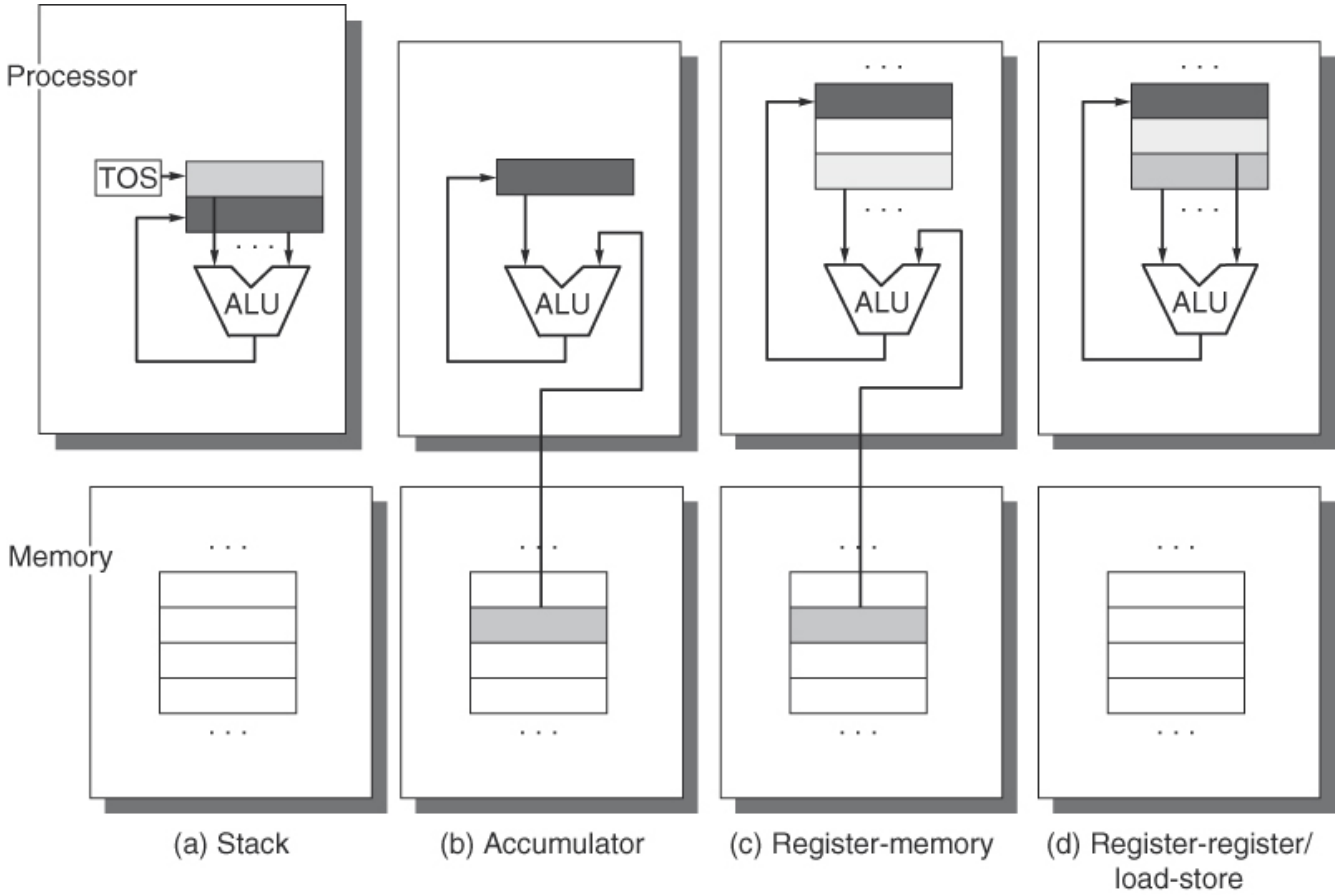
- Computation instructions
  - Integer arithmetic/logic instructions, e.g., add, and, mult
  - Floating-point instructions
- Memory transfer instructions
  - Loads/Stores
- Control instructions
  - Conditional branch
  - Unconditional jump

Number of Memory Operands in ALU Ops

- Consider a HLL statement C<- A + B

- With 3 memory operands (memory-to-memory instruction)
  - ADD C,A,B
- With 1 memory operand and 1 register operand
  - LW R1,A
  - ADD R1,B
  - SW R1,C
- With no memory operand (Load/Store architecture)
  - LW R1,A
  - LW R2,B
  - ADD R3,R1,R2
  - SW R3,C

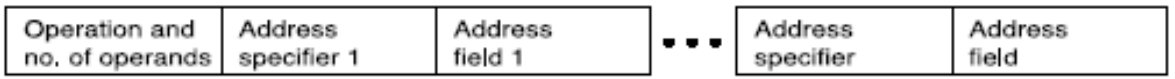
ISA Classification based on Operand Locations



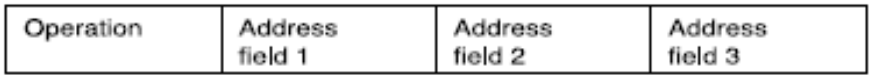
Virtually every new architecture designed after 1980 uses a load-store register architecture! Why?

ISA Classification based on Encoding

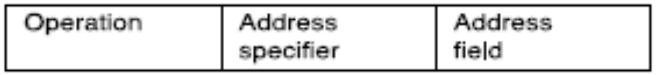
- ISA encoding determines code size and decoding complexity
  - Decoding is simplified if instruction format is highly predictable

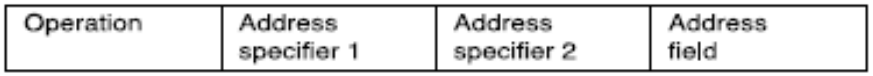


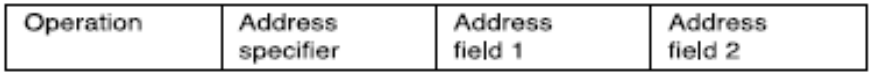
(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







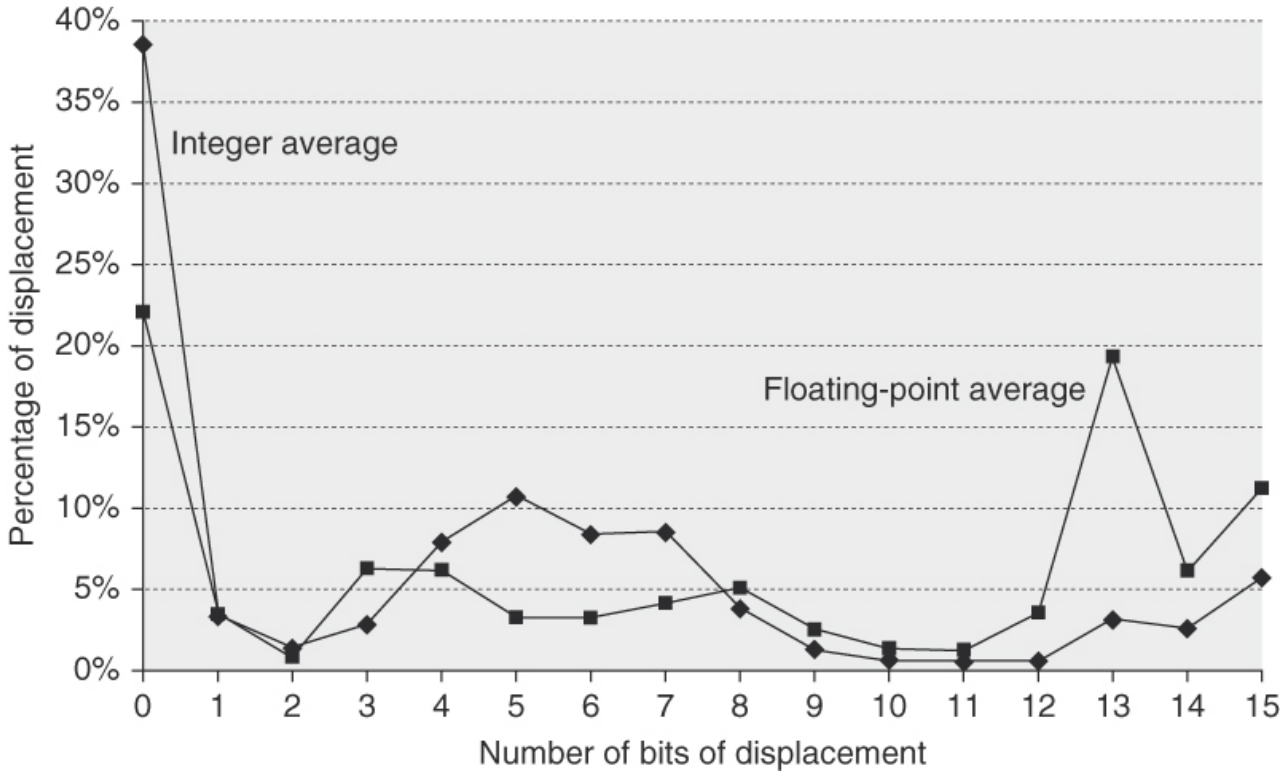
(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Addressing Modes

MODE	EXAMPLE	MEANING
REGISTER	ADD R4,R3	reg[R4] <- reg[R4] +reg[R3]
IMMEDIATE	ADD R4, #3	reg[R4] <- reg[R4] + 3
DISPLACEMENT	ADD R4, 100(R1)	reg[R4] <- reg[R4] + Mem[100 + reg[R1]]
REGISTER INDIRECT	ADD R4, (R1)	reg[R4] <- reg[R4] + Mem[reg[R1]]
INDEXED	ADD R3, (R1+R2)	reg[R3] <- reg[R3] + Mem[reg[R1] + reg[R2]]
DIRECT OR ABSOLUTE	ADD R1, (1001)	reg[R1] <- reg[R1] + Mem[1001]
MEMORY INDIRECT	ADD R1, @R3	reg[R1] <- reg[R1] + Mem[Mem[Reg[3]]]
POST INCREMENT	ADD R1, (R2)+	ADD R1, (R2) then R2 <- R2+d
PREDECREMENT	ADD R1, -(R2)	R2 <- R2-d then ADD R1, (R2)
PC-RELATIVE	BEZ R1, 100	if R1==0, PC <- PC+100
PC-RELATIVE	JUMP 200	Concatenate bits of PC and offset

Actual Use of Addressing Modes

- Displacement and Immediate are the most common addressing modes
  - 16 bits is usually enough for both types of values



Actual Use of Addressing Modes

- More complex addressing modes can be synthesized
  - Memory indirect: LW R1, @(R2)
    - » LW R3, 0(R2)
    - » LW R1, 0(R3)
  - Post increment: LW R1, (R2)++
    - » LW R1, 0(R2)
    - » ADDI R2, R2, #size

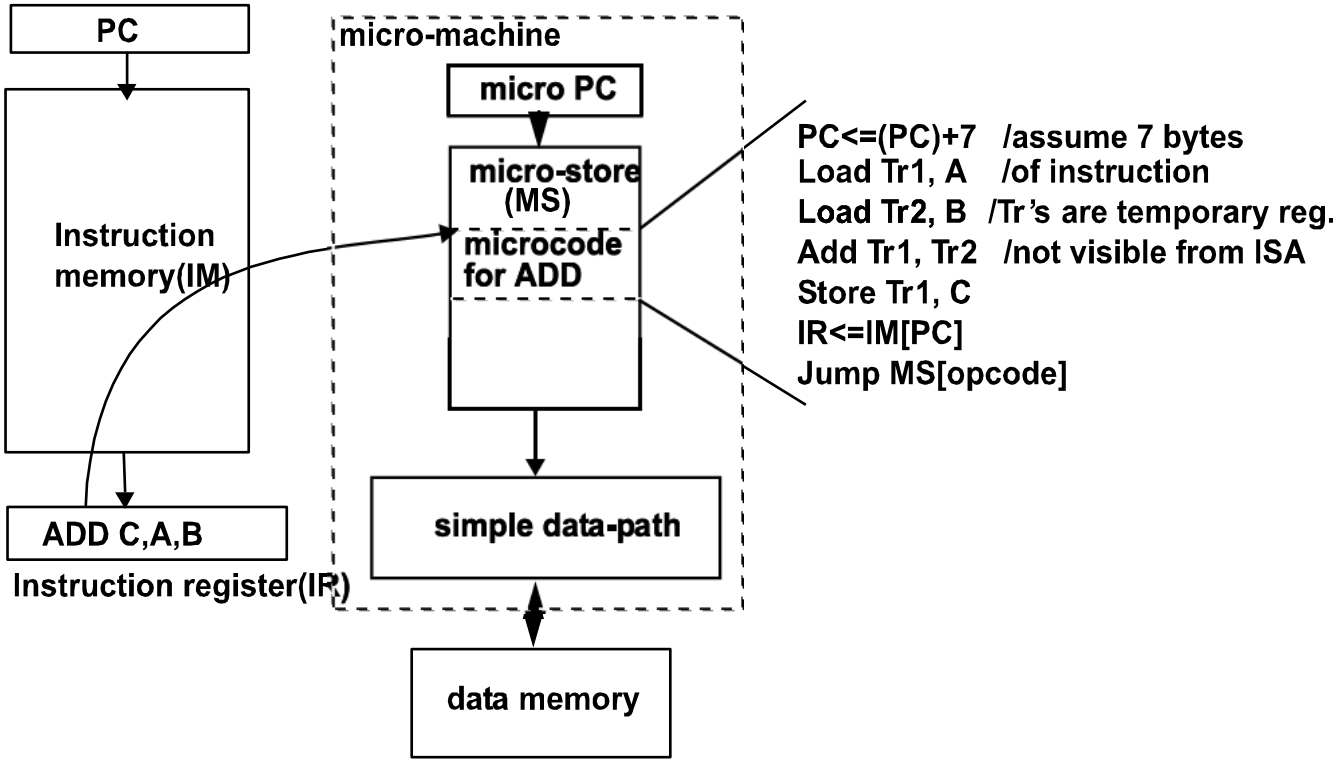
RISC vs. CISC

- Complex vs. Reduced Instruction Set Computers

- RISC design philosophy
  - load-store architecture
  - Fixed/hybrid instruction lengths
  - limited addressing modes
  - limited operations

RISC vs. CISC

- The definition of CISC/RISC is not directly related to implementation!
- Today's CISC machines are usually implemented internally as RISC using microcode, with some translation overhead



Important ISAs and Implementations

ISA	Company	Implementations	Type
System 370	IBM	IBM 370/3081	CISC--Legacy
x86	Intel	Intel 386, Intel Pentium, AMD Turion	CISC-Legacy
Motorola68000	Motorola	Motorola 68020	CISC-Legacy
Sun SPARC	Sun Microsystems	SPARC T2	RISC
PowerPC	IBM/Motorola	PowerPC-6	RISC
ARM	ARM	ARM Coretex A8 Apple A5/A6/A7 Qualcomm Snapdragon NVida Tegra ...	RISC
MIPS	MIPS/SGI	MIPS10000	RISC
IA-64	Intel	Itanium-2	RISC

Core ISA Used in This Course

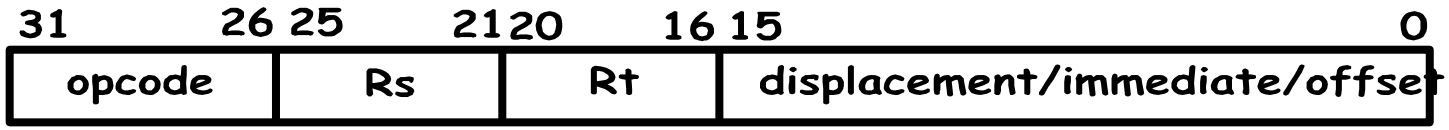
Types	Opcode	Assembly code	Meaning	Comments
Data Transfers	LB, LH, LW, LD	LW R1,#20(R2)	R1<=MEM[(R2)+20]	for bytes, half-words
	SB, SH, SW, SD	SW R1,#20(R2)	MEM[(R2)+20]<=(R1)	words, and double words
	L.S, L.D	L.S F0,#20(R2)	F0<=MEM[(R2)+20]	single/double float load
	S.S, S.D	S.S F0,#20(R2)	MEM[(R2)+20]<=(F0)	single/double float store
ALU operations	ADD, SUB, ADDU, SUBU	ADD R1,R2,R3	R1<=(R2)+(R3)	add/sub signed or unsigned
	ADDI, SUBI, ADDIU, SUBIU	ADDI R1,R2,#3	R1<=(R2)+3	add/sub immediate signed or unsigned
	AND, OR, XOR,	AND R1,R2,R3	R1<=(R2).AND.(R3)	bitwise logical AND, OR, XOR
	ANDI, ORI, XORI,	ANDI R1,R2,#4	R1<=(R2).ANDI.4	bitwise AND, OR, XOR immediate
	SLT, SLTU	SLT R1,R2,R3	R1<=1 if R2<R3 else R1<=0	test on R2,R3 outcome in R1, signed or unsigned comparison
	SLTI, SLTUI	SLTI R1,R2,#4	R1<=1 if R2<4 else R1<=0	test R2 outcome in R1, signed or unsigned comparison

Core ISA Used in This Course

Types	Opcode	Assembly code	Meaning	Comments
Branches/Jumps	BEQZ, BNEZ	BEQZ R1,label	PC<=label if (R1)=0	conditional branch-equal 0/not equal 0
	BEQ, BNE	BNE R1,R2,label	PC<=label if (R1)=(R2)	conditional branch-equal/not equal
	J	J target	PC<=target	target is an immediate field
	JR	JR R1	PC<=(R1)	target is in register
	JAL	JAL target	R1<=(PC)+4; PC<=target	jump to target after saving the return address in R31
Floating point	ADD.S,SUB.S,MUL.S, DIV.S	ADD.S F1,F2,F3	F1<=(F2)+(F3)	float arithmetic single precision
	ADD.D,SUB.D,MUL.D, DIV.D	ADD.D F0,F2,F4	F0<=(F2)+(F4)	float arithmetic double precision

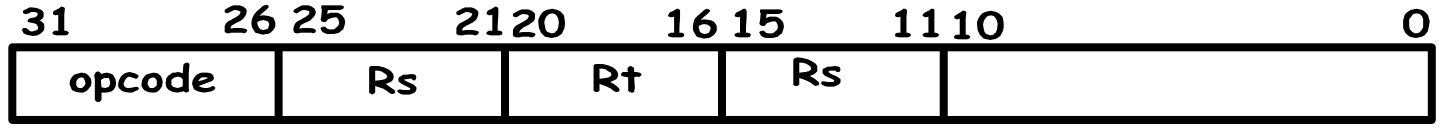
Instruction Formats

I-type



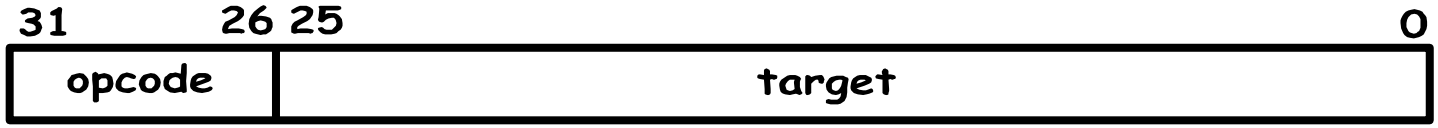
LW Rt, displacement(Rs)
SW Rt, displacement(Rs)
ADDI Rt, Rs, immediate
BEQ Rt, Rs, offset

R-type



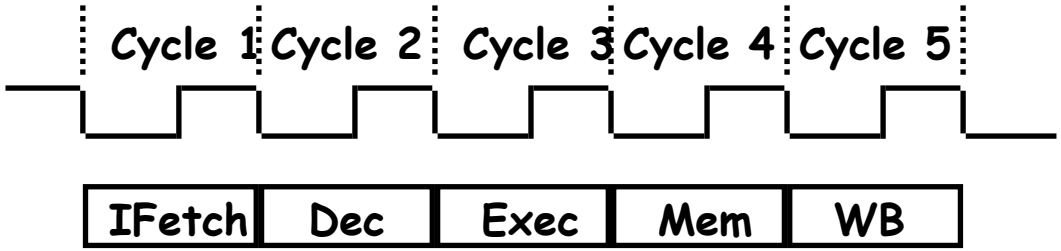
ADD Rd, Rt, Rs

J-type



J target
JAL target

The Five Instruction Execution Steps

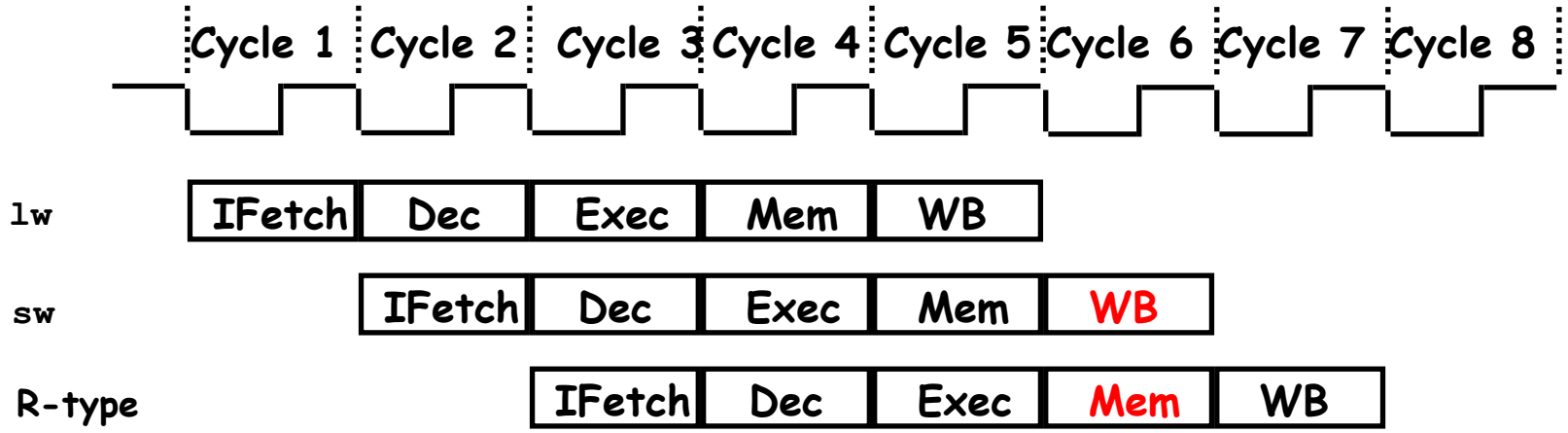


- IFetch: Instruction Fetch and Update PC
- Dec: Instruction Decode, Register Read, Sign Extend Offset
- Exec: Execute R-type; Calculate Memory Address; Branch Comparison; Branch and Jump Completion
- Mem: Memory Read; Memory Write Completion; R-type Completion (RegFile write)
- WB: Memory Read Completion (RegFile write)

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

A Pipelined MIPS Processor

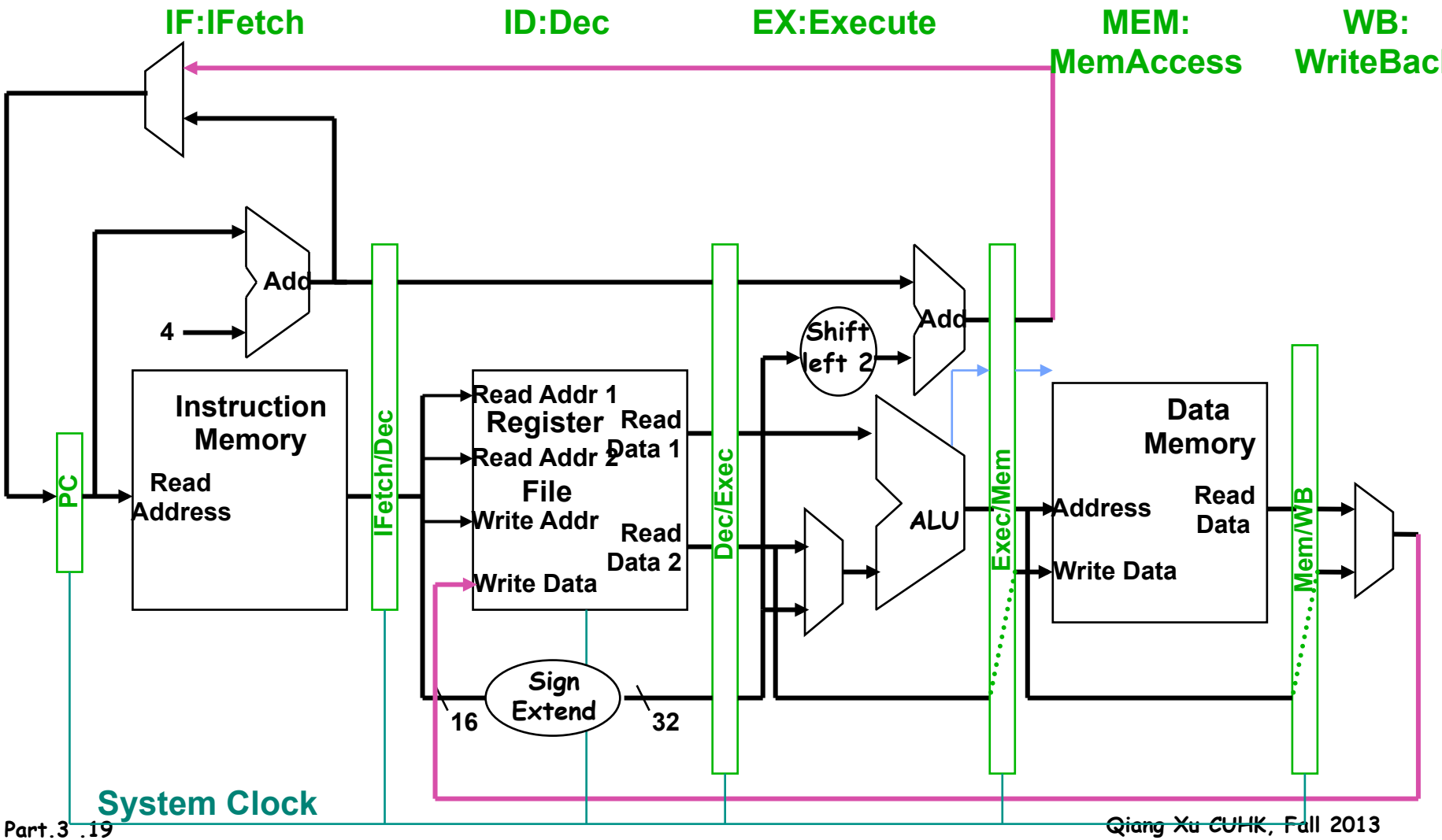
- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is not reduced



  - » clock cycle (pipeline stage time) is limited by the slowest stage
  - » for some instructions, some stages are wasted cycles

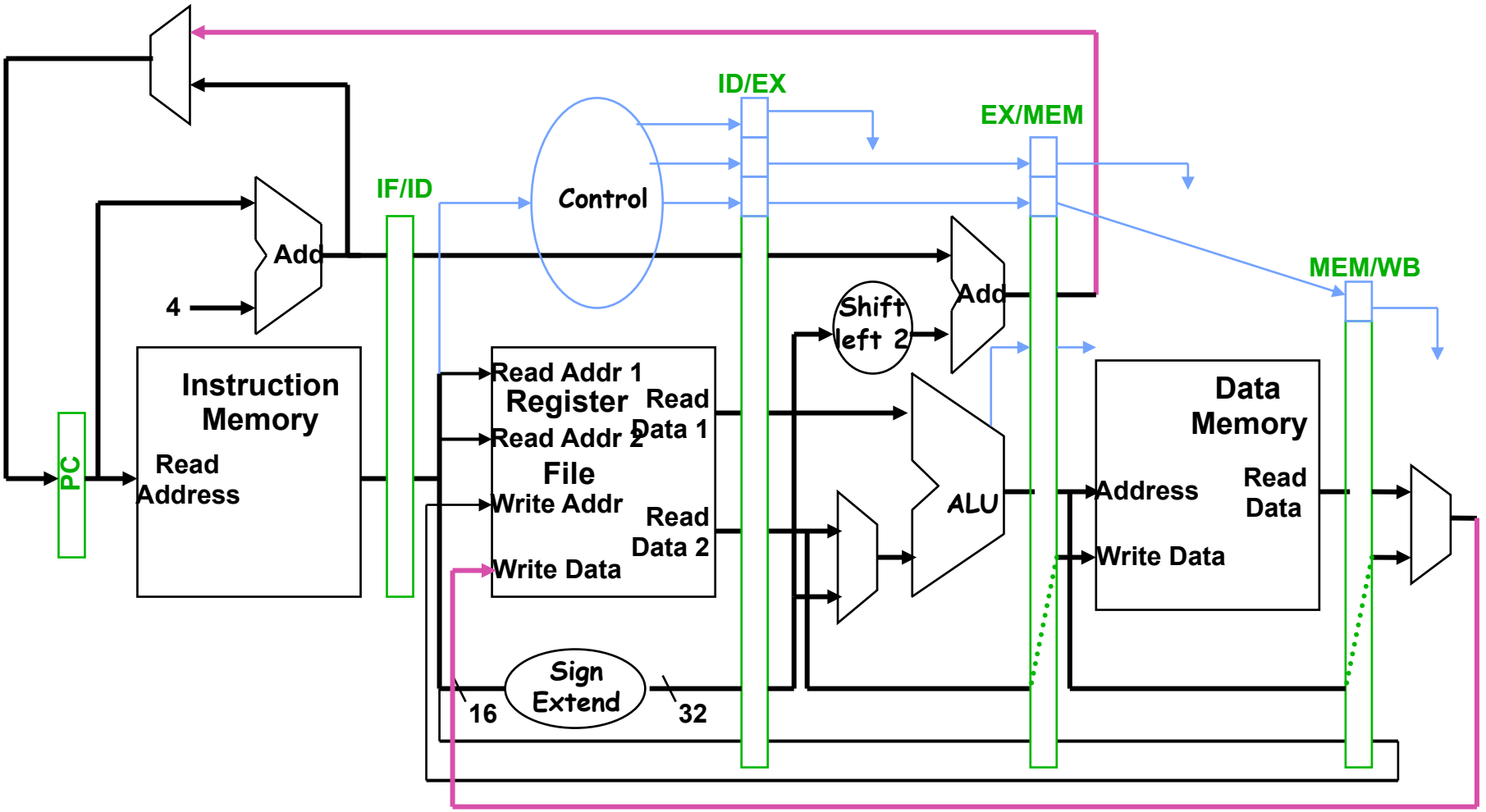
MIPS Pipeline Datapath

- Require State registers between each pipeline stage to isolate them



MIPS Pipeline Control Path

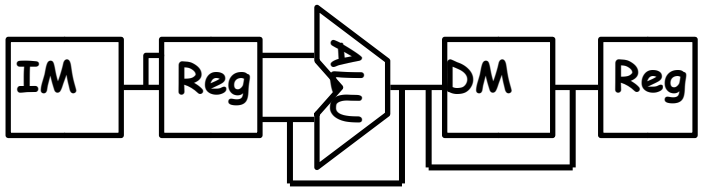
- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages



Pipelining the MIPS ISA

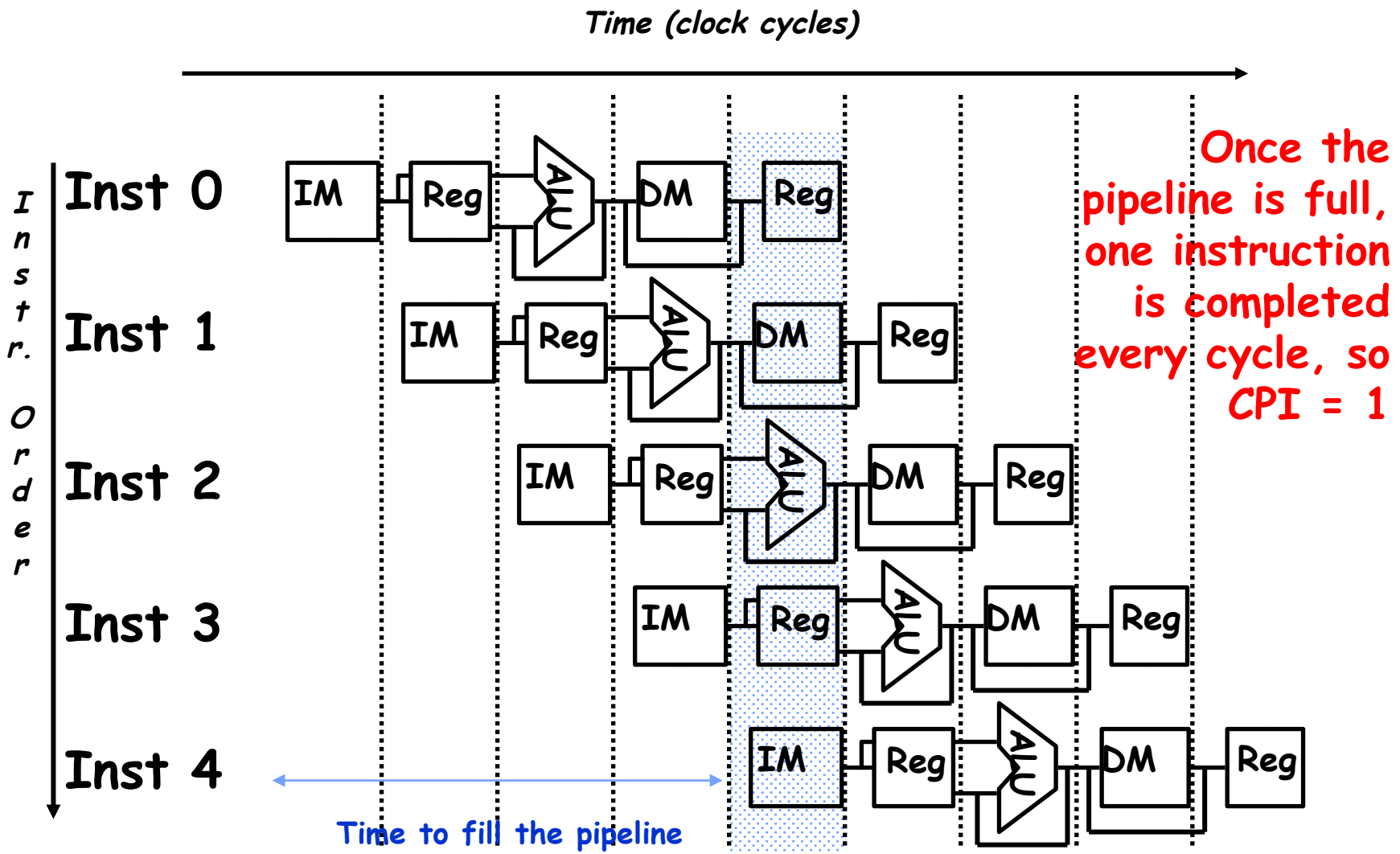
- **What makes it easy**
  - **all instructions are the same length (32 bits)**
    - » can fetch in the 1st stage and decode in the 2nd stage
  - **few instruction formats (three) with symmetry across formats**
    - » can begin reading register file in 2nd stage
  - **memory operations can occur only in loads and stores**
    - » can use the execute stage to calculate memory addresses
  - **each MIPS instruction writes at most one result (i.e., changes the machine state) and does so near the end of the pipeline (MEM and WB)**

Graphically Representing MIPS Pipeline



- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!

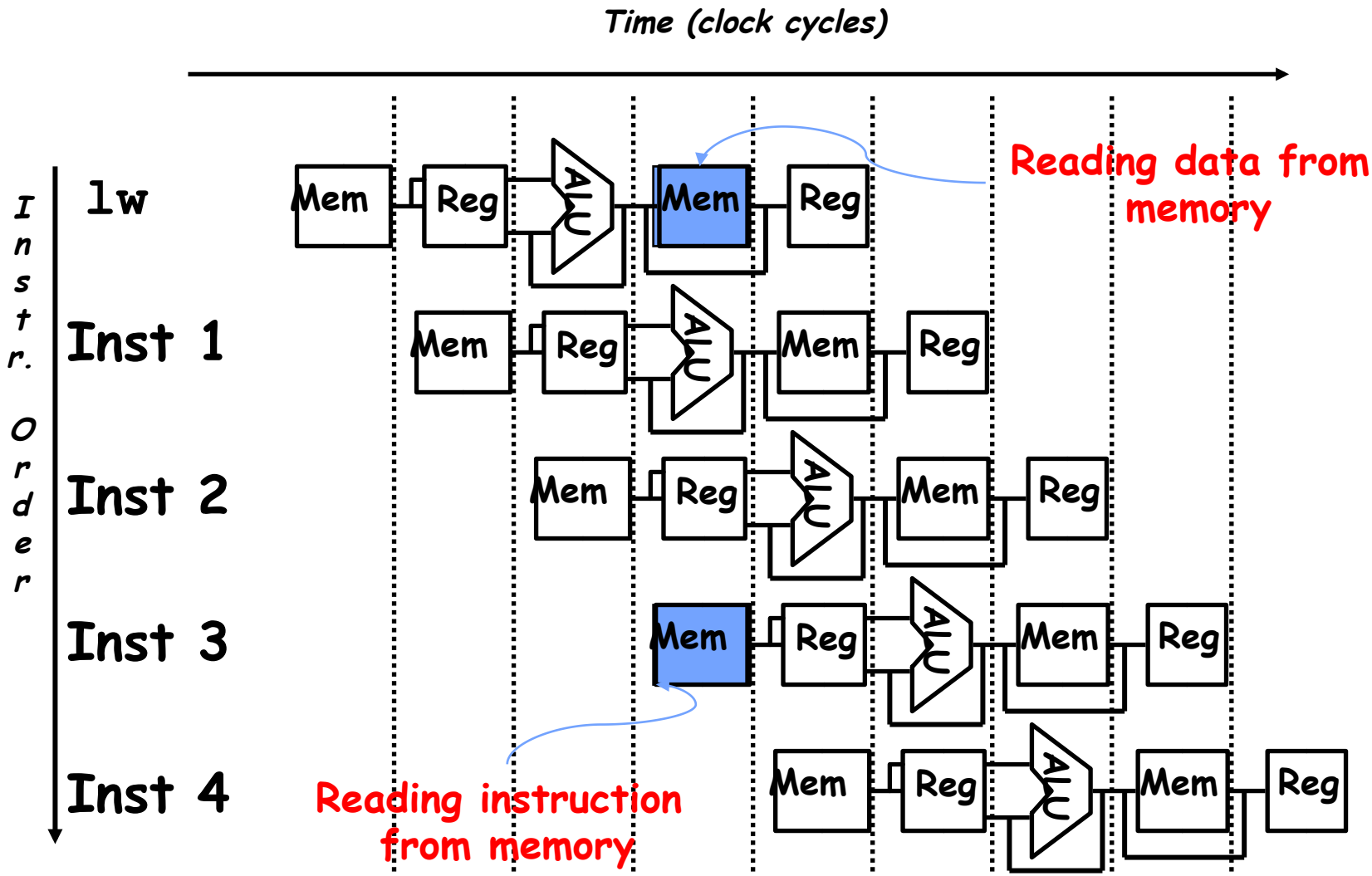


Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - » An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - » branch instructions

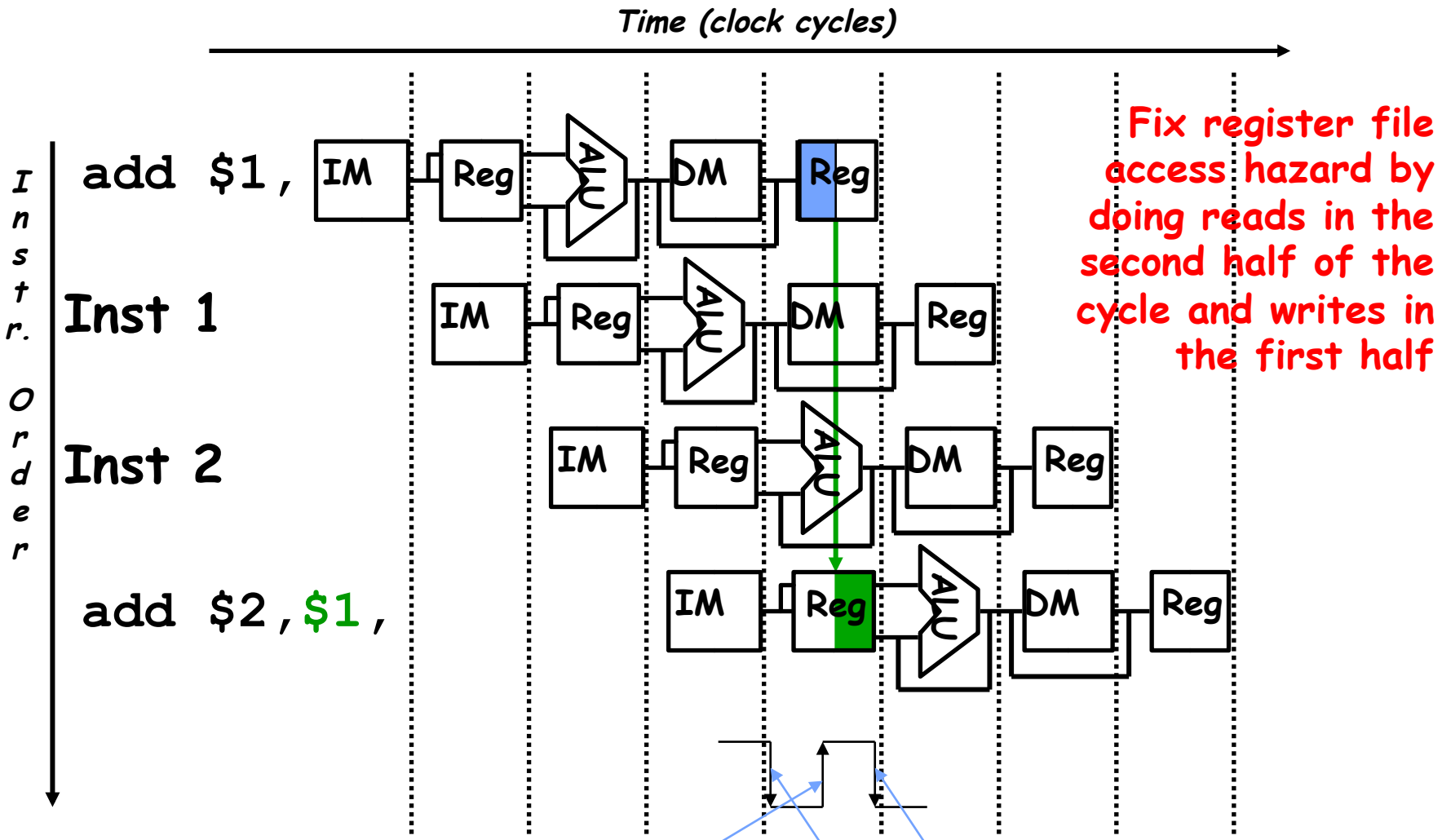
- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

A Single Memory Would Be a Structural Hazard



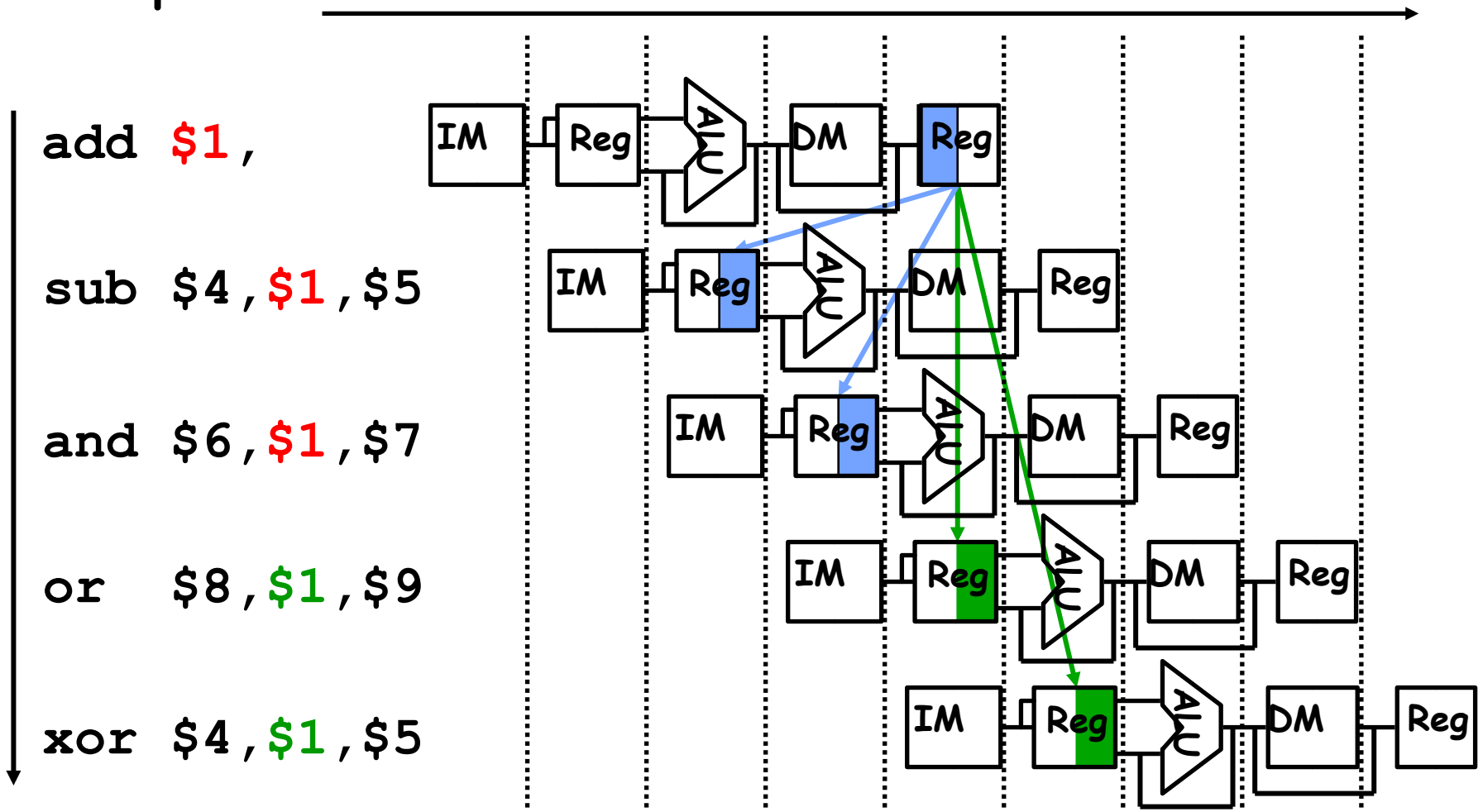
Fix with separate instr and data caches (I$ and D$)

How About Register File Access?



Register Usage Can Cause Data Hazards

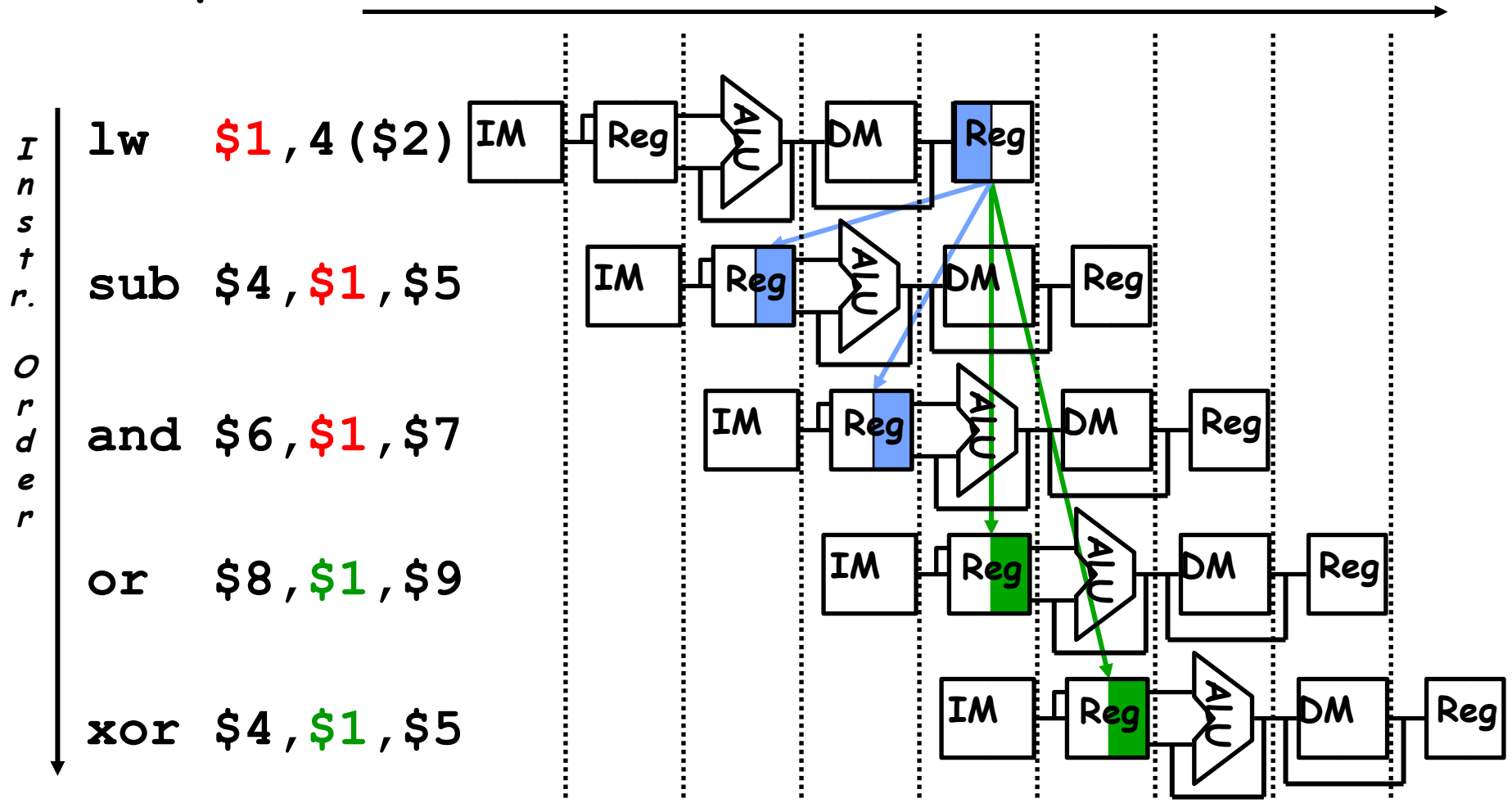
- Dependencies backward in time cause hazards



Read before write data hazard

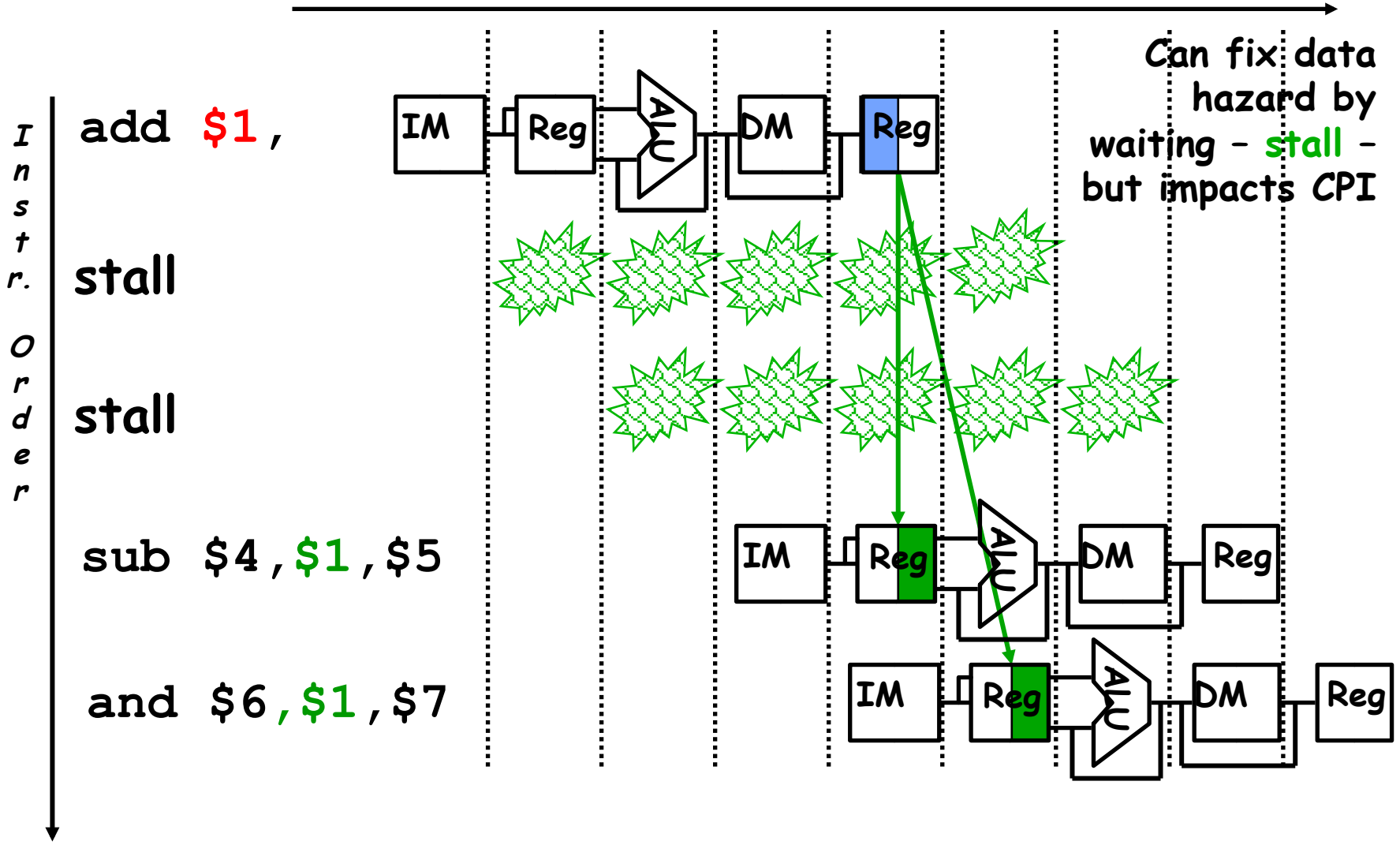
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

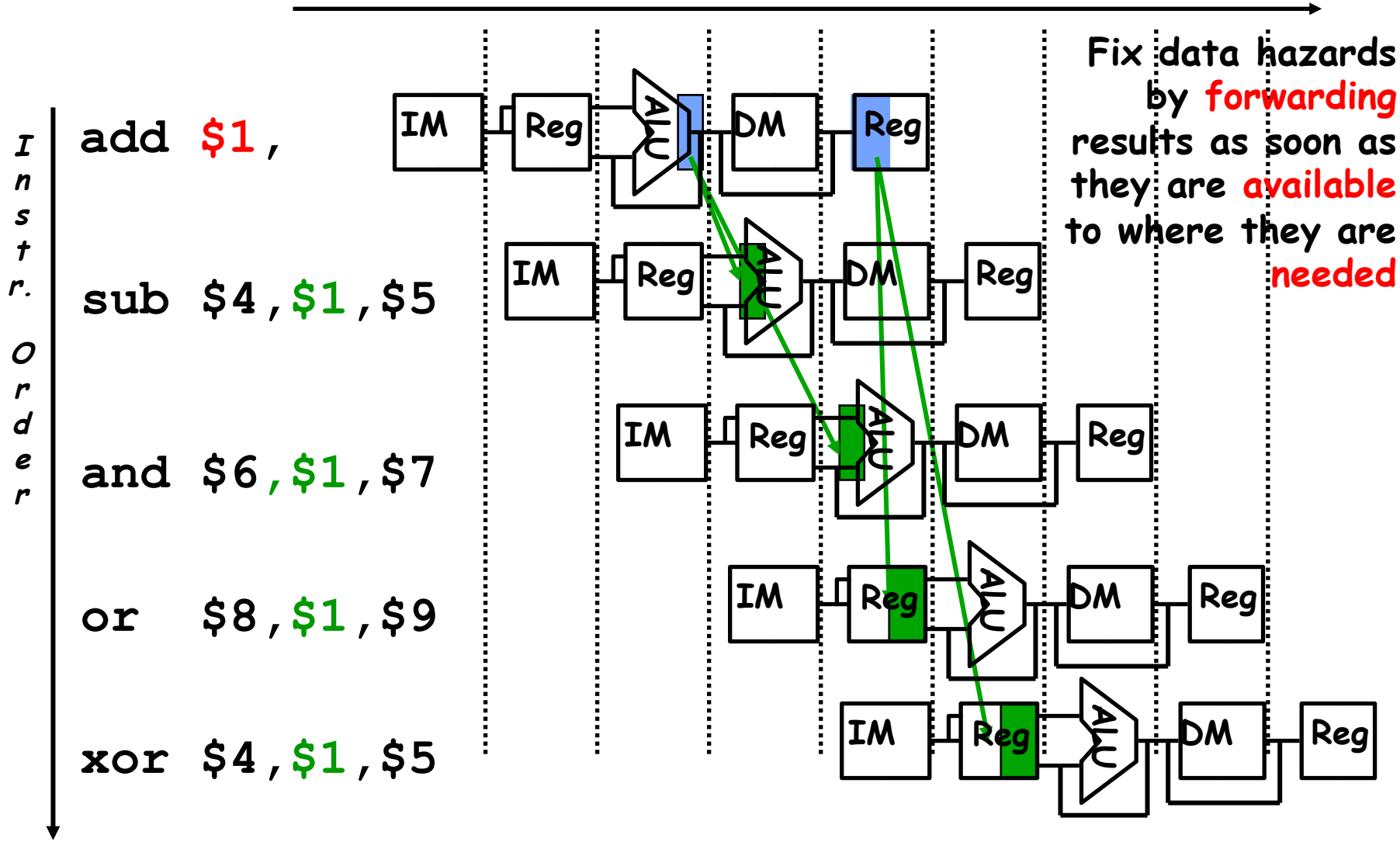


Load-use data hazard

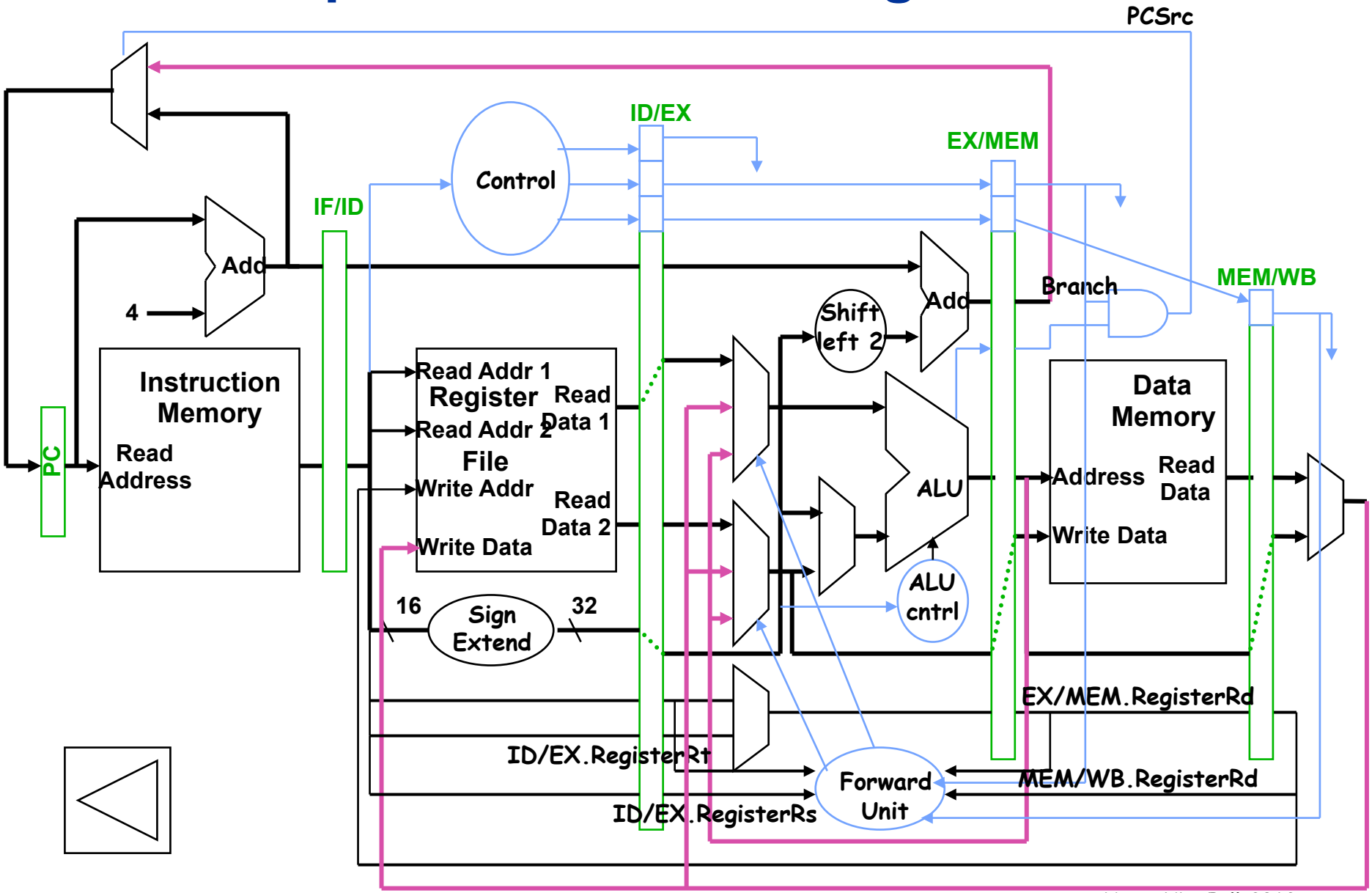
One Way to "Fix" a Data Hazard



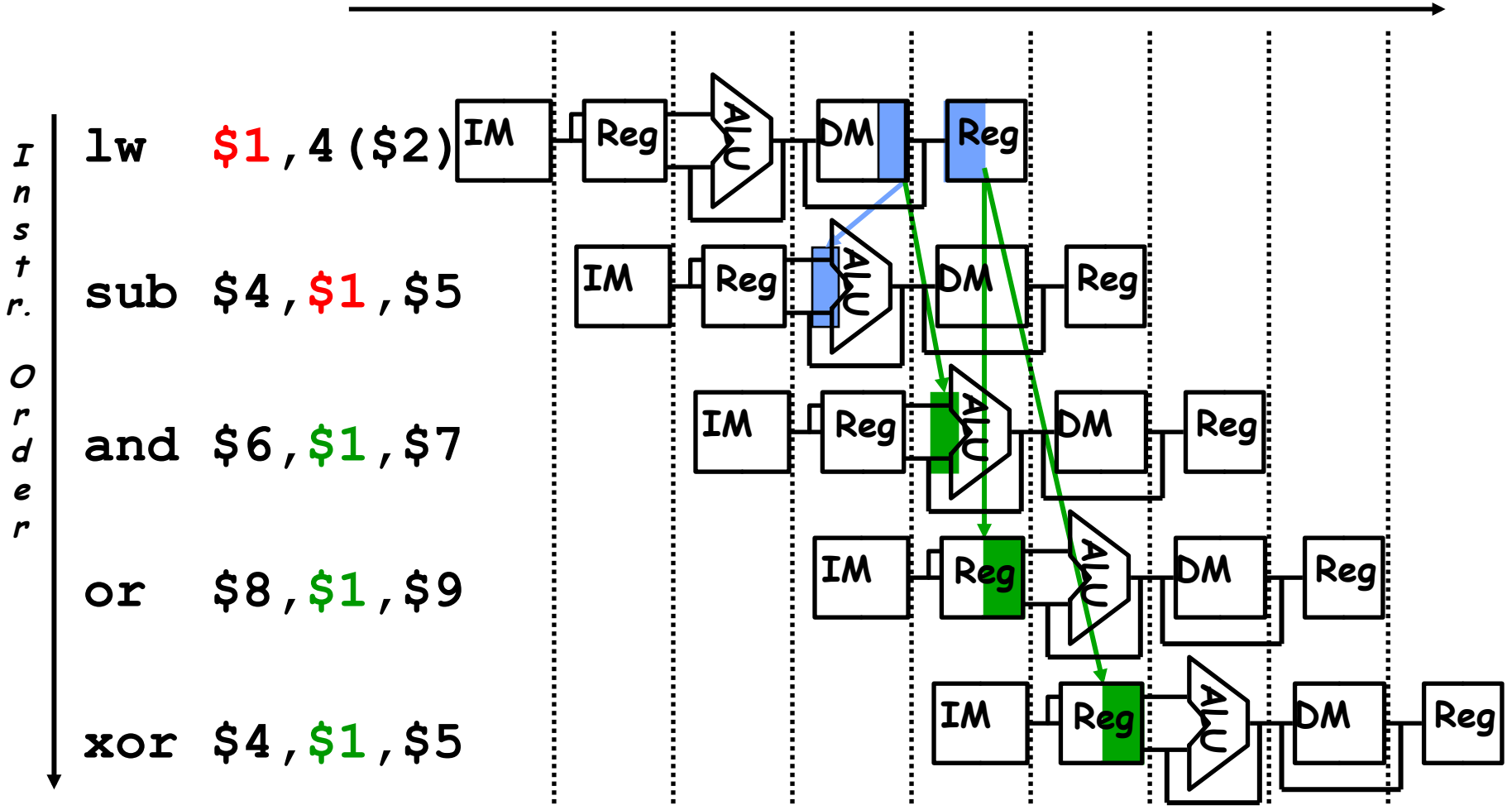
Another Way to "Fix" a Data Hazard



Datapath with Forwarding Hardware

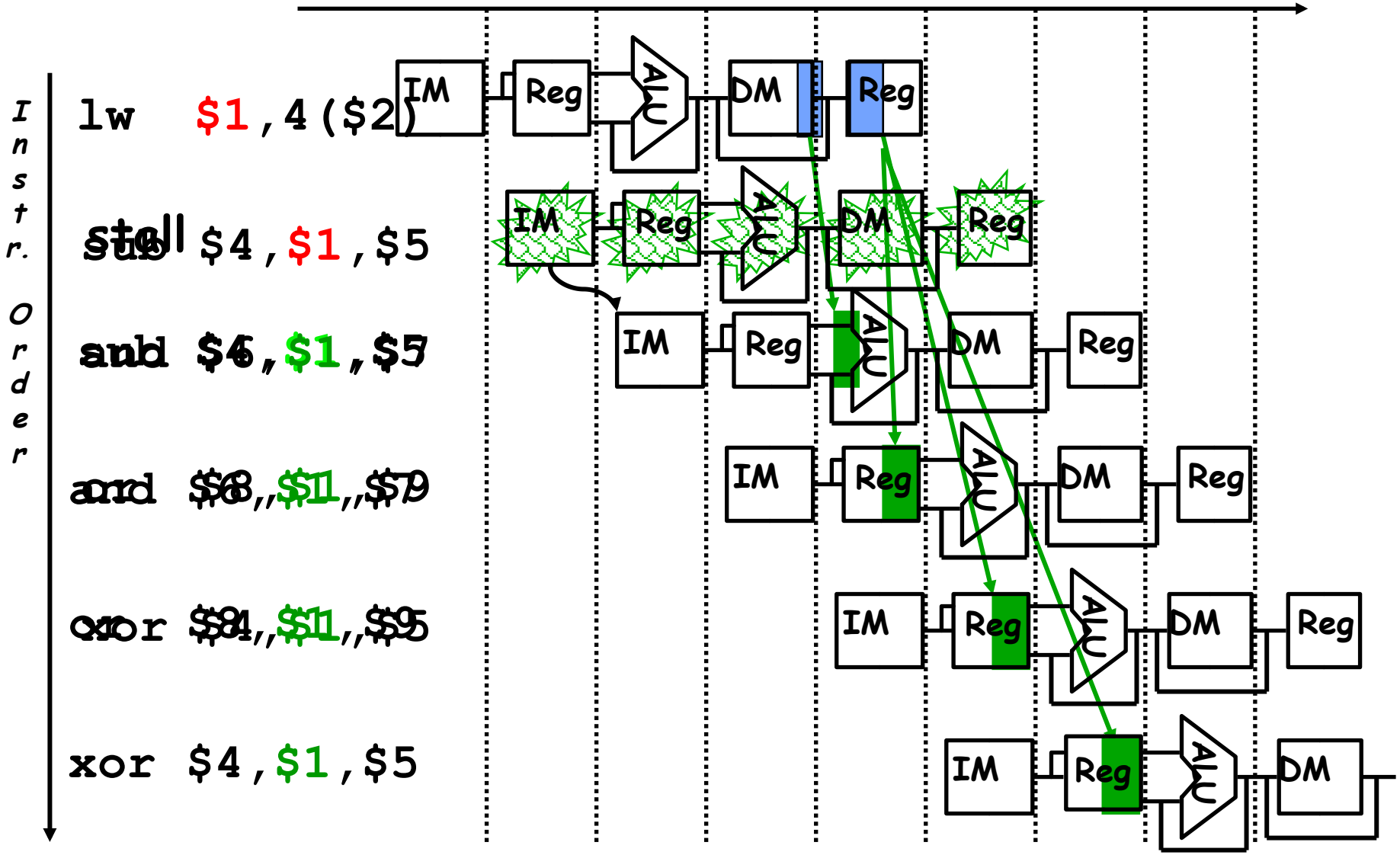


Forwarding with Load-use Data Hazards



- Will still need one stall cycle even with forwarding

Forwarding with Load-use Data Hazards

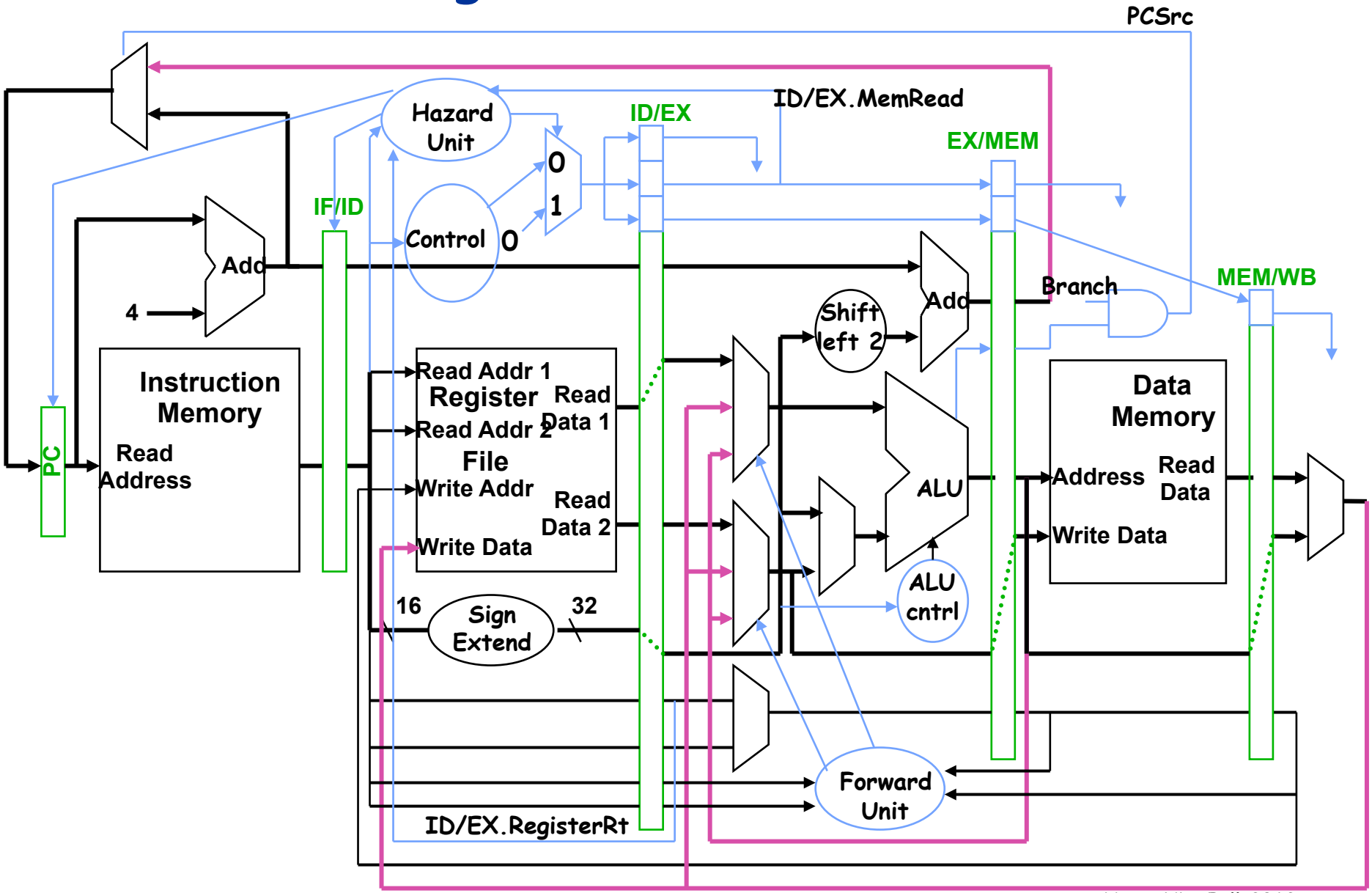


Load-use Hazard Detection Unit

- Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

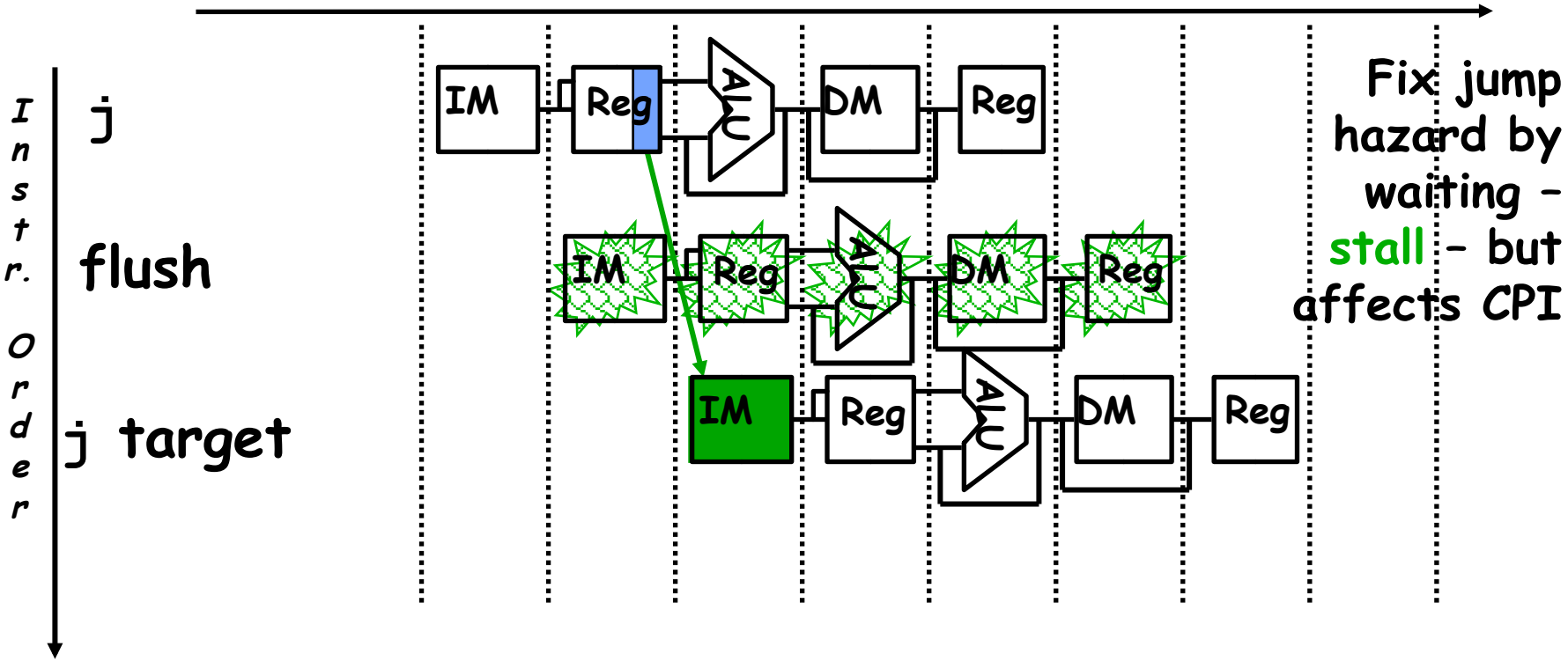
- Insert a "bubble" between the lw instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a noop in the execution stream)
  - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (noop). The Hazard Unit controls the mux that chooses between the real control values and the 0's.

Adding the Hazard Hardware



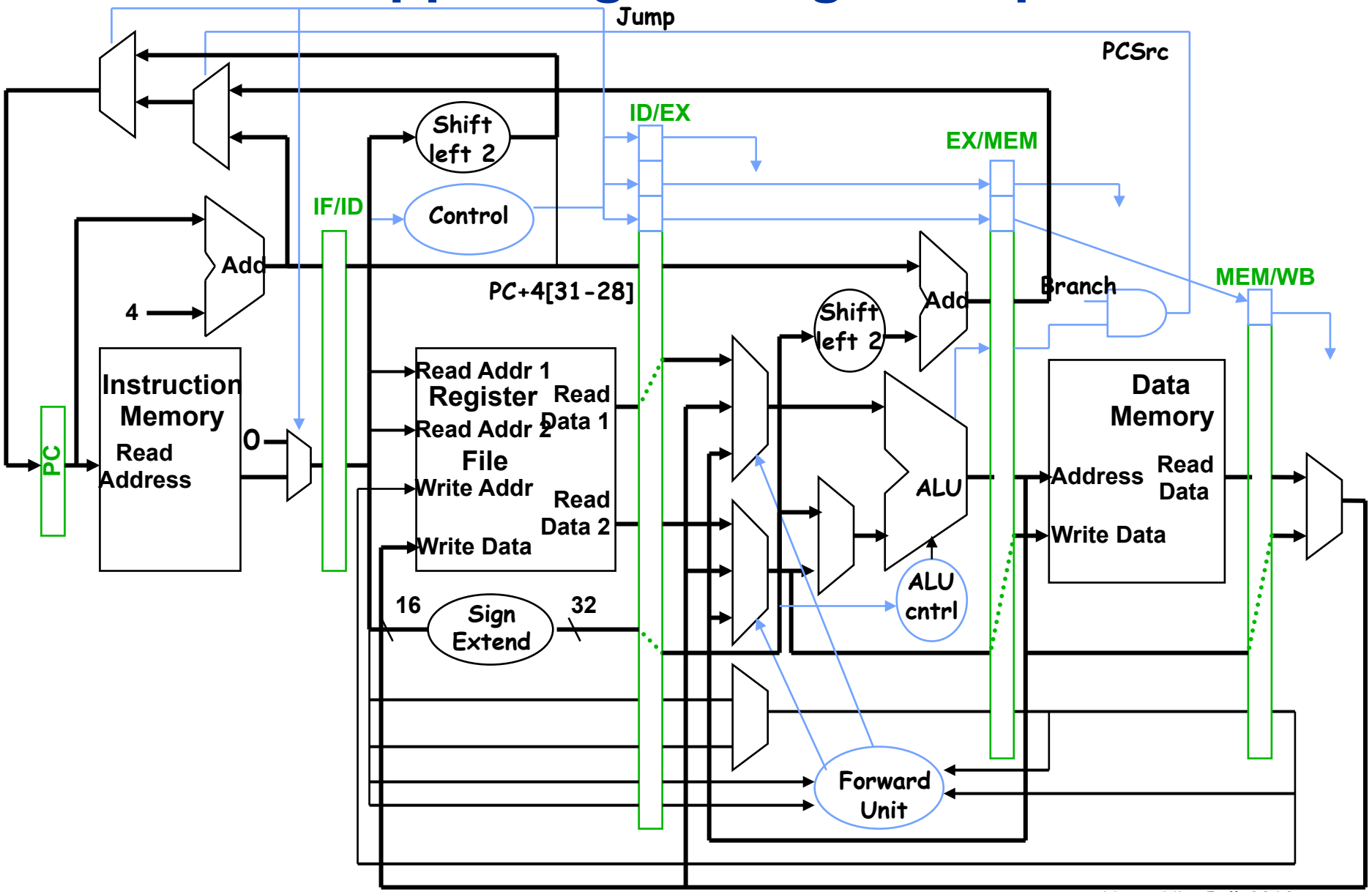
Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed



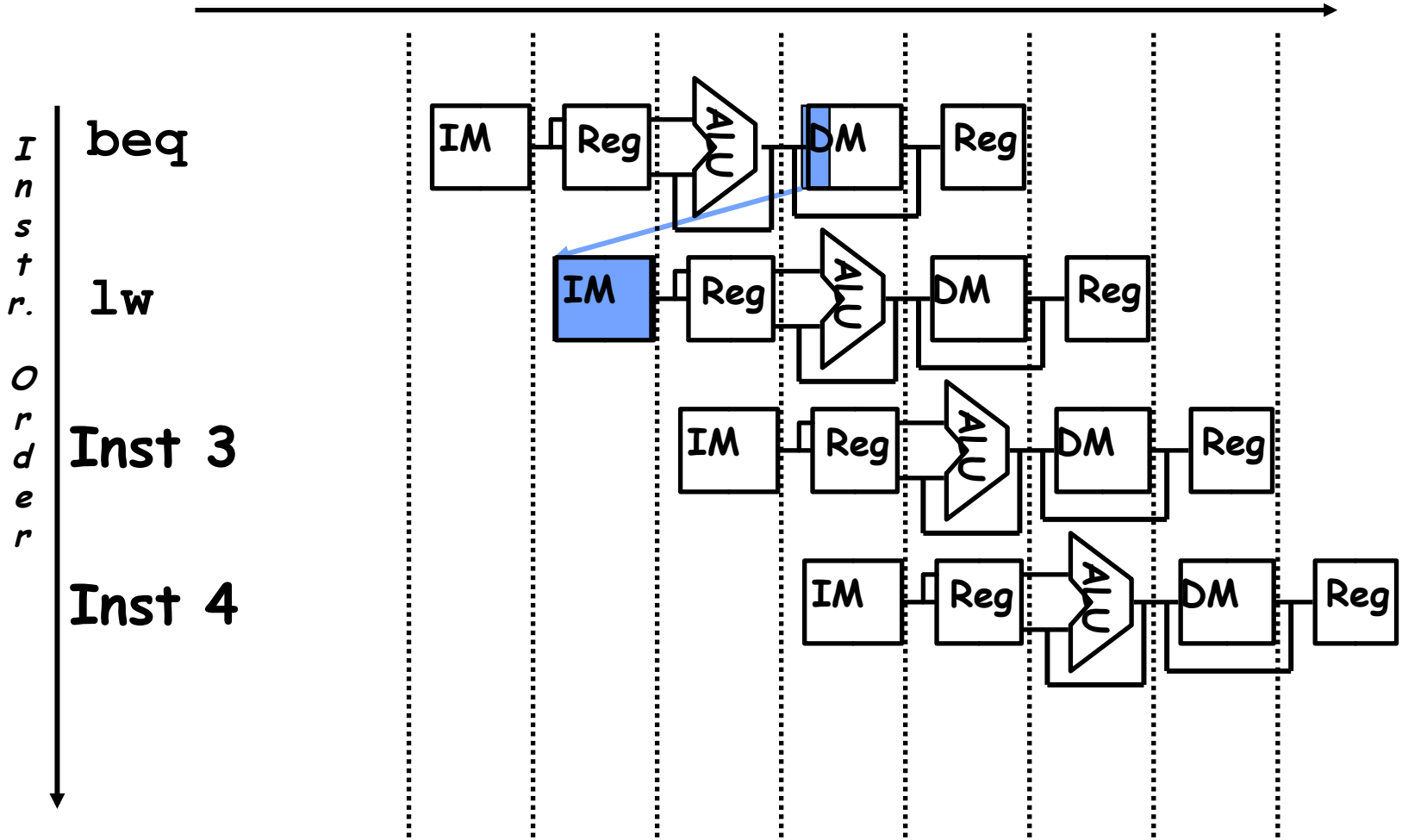
- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Supporting ID Stage Jumps

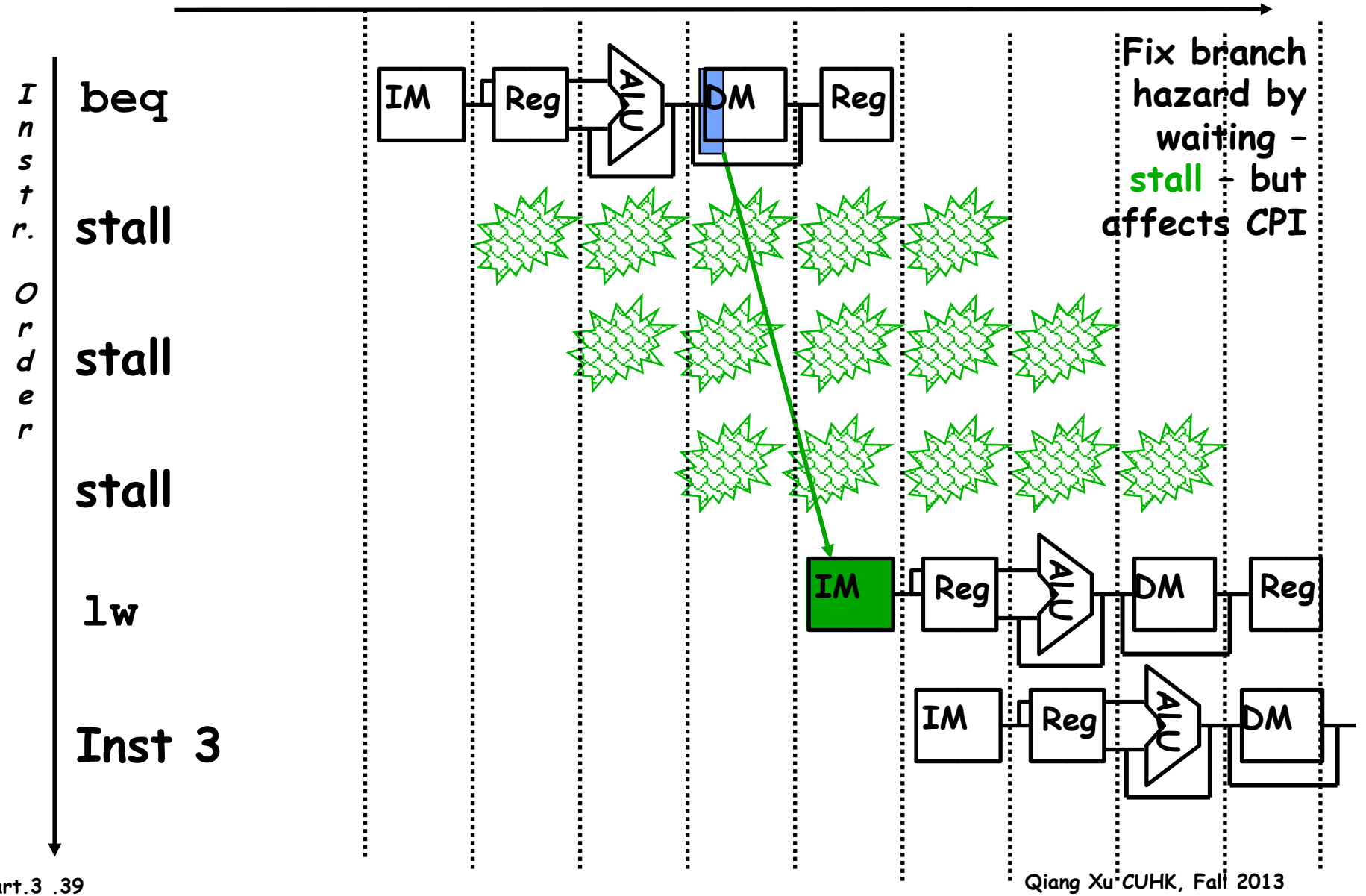


Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



One Way to "Fix" a Control Hazard



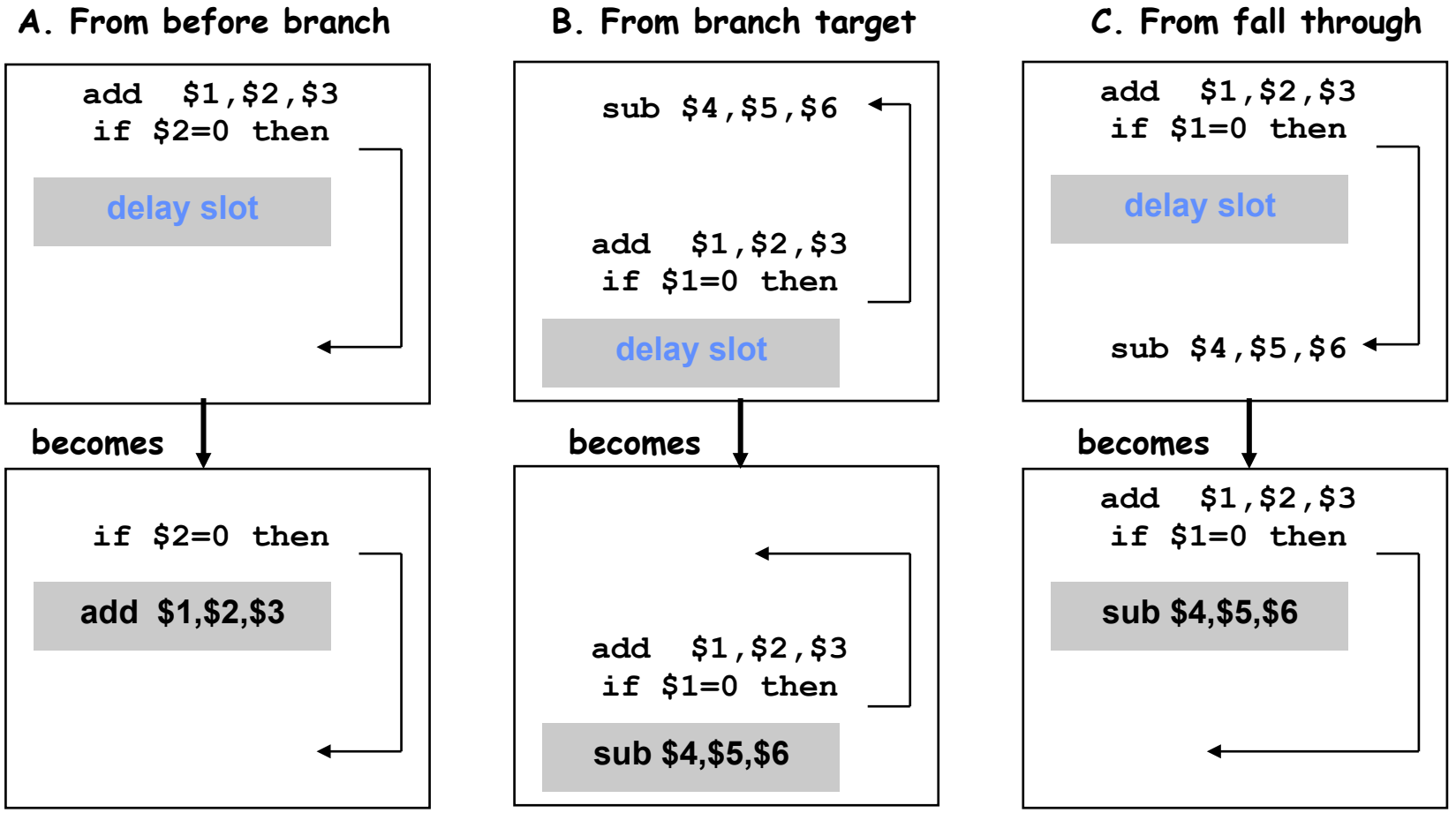
Moving Branch Decisions Earlier in Pipe

- Move the branch decision hardware back to the EX stage
  - Reduces the number of stall (flush) cycles to two
  - Adds an and gate and a 2x1 mux to the EX timing path
- Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  - Reduces the number of stall (flush) cycles to one (like with jumps)
    - But now need to add forwarding hardware in ID stage
  - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an and gate to the ID timing path
- For deeper pipelines, branch decision points can be even later in the pipeline, incurring more stalls

Delayed Decision

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect after that next instruction
  - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots

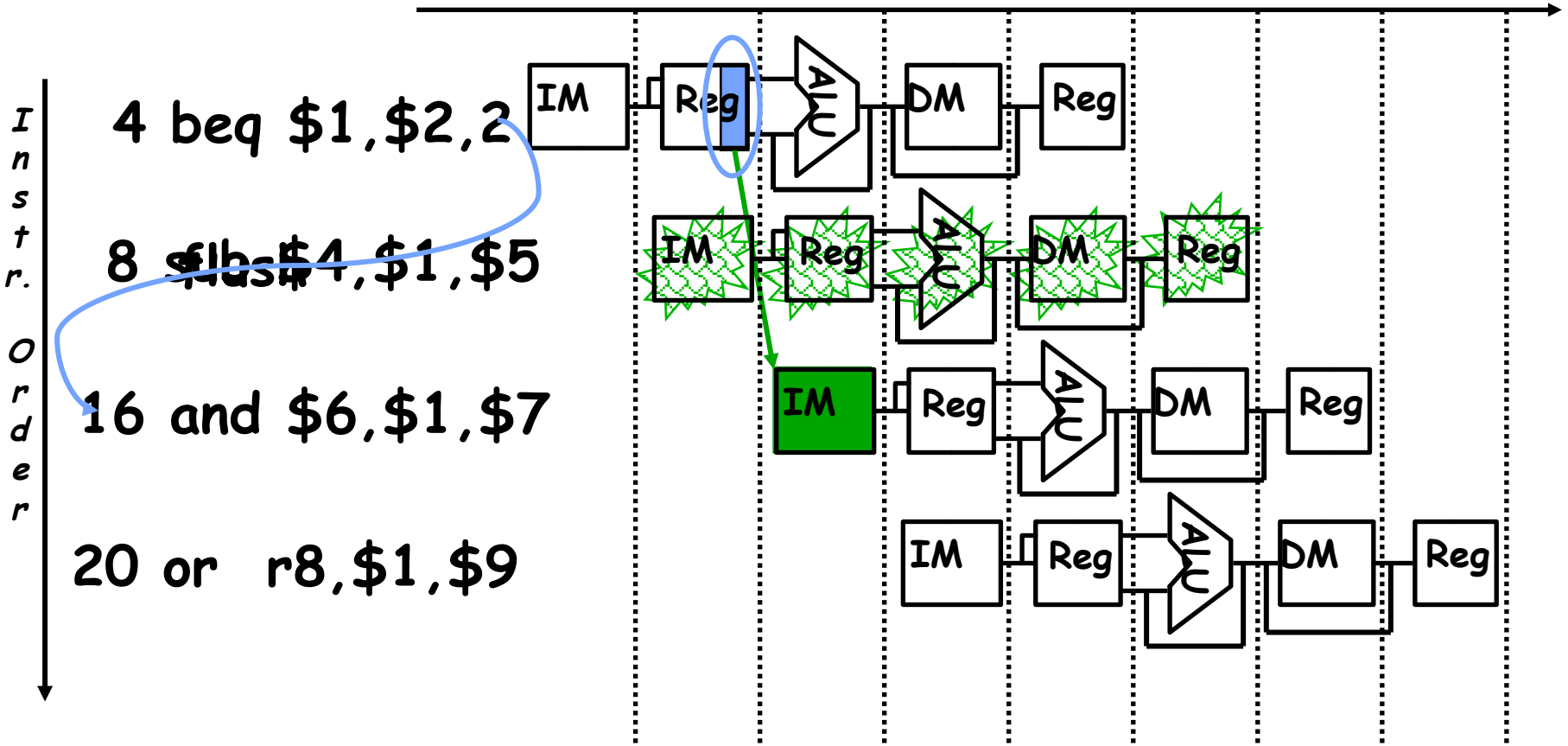


- A is the best choice, fills delay slot and reduces IC
- In B and C, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
- Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch is taken does the pipeline stall
  - If taken, flush instructions after the branch (earlier in the pipeline)
    - » in IF, ID, and EX stages if branch logic in MEM – three stalls
    - » In IF and ID stages if branch logic in EX – two stalls
    - » in IF stage if branch logic in ID – one stall
  - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
  - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert IF.Flush to zero the instruction field of the IF/ID pipeline register (transforming it into a noop)

Branching Structures

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
          j  Loop
Out:  fall out instr
```

- Predict not taken doesn't work well for "bottom of the loop" branching structures

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

Static Branch Prediction, con't

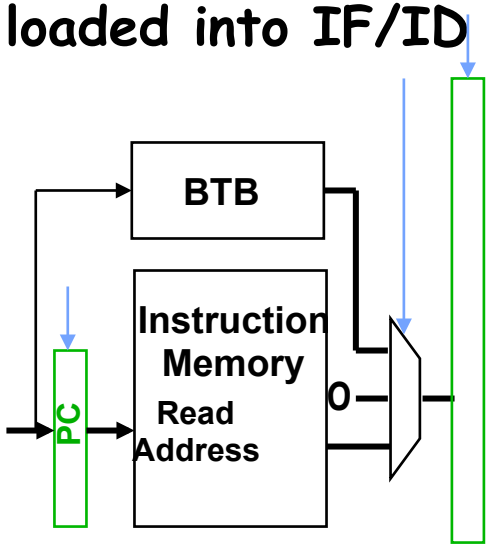
- Resolve branch hazards by assuming a given outcome and proceeding
- Predict taken – predict branches will always be taken
  - Predict taken always incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
  - Is there a way to "cache" the address of the branch target instruction ??
- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution
- Dynamic branch prediction – predict branches at run-time using run-time information

Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains a bit passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
    - » Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit
    - » A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

Branch Target Buffer

- The BHT predicts when a branch is taken, but does not tell where its taken to!
  - A branch target buffer (BTB) in the IF stage can cache the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which "next" instruction will be loaded into IF/ID at the next clock edge
    - » Would need a two read port instruction memory
  - Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction



- If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

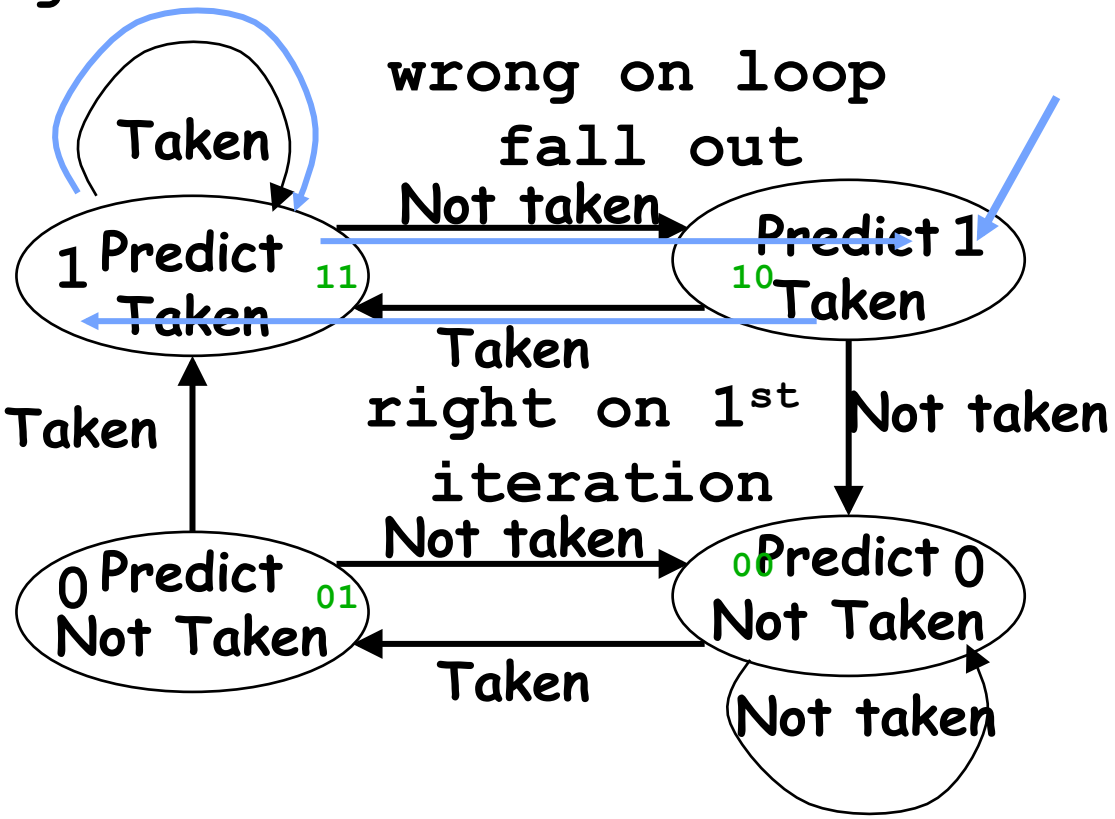
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  - First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  - As long as branch is taken (looping), prediction is correct
  - Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
                .
                .
                .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed right 9 times




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
last loop instr
bne $1,$2,Loop
fall out instr
```

- BHT also stores the initial FSM state

Dealing with Exceptions

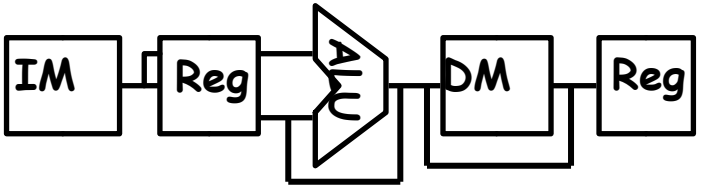
- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and "deals" with it

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler
  - simply suspend and resume user program

- Traps – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so much stop the offending instruction midstream in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

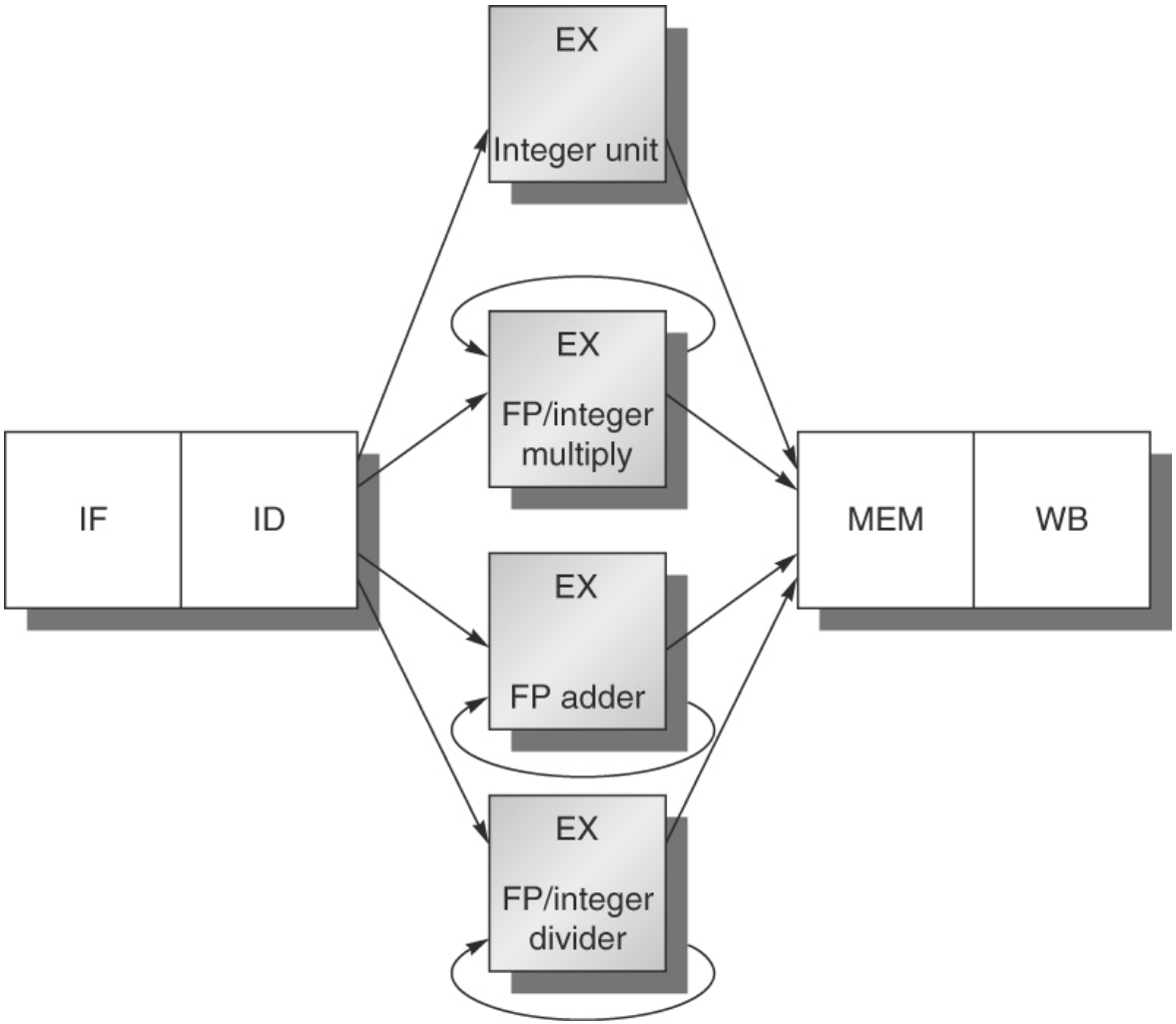
Where in the Pipeline Exceptions Occur



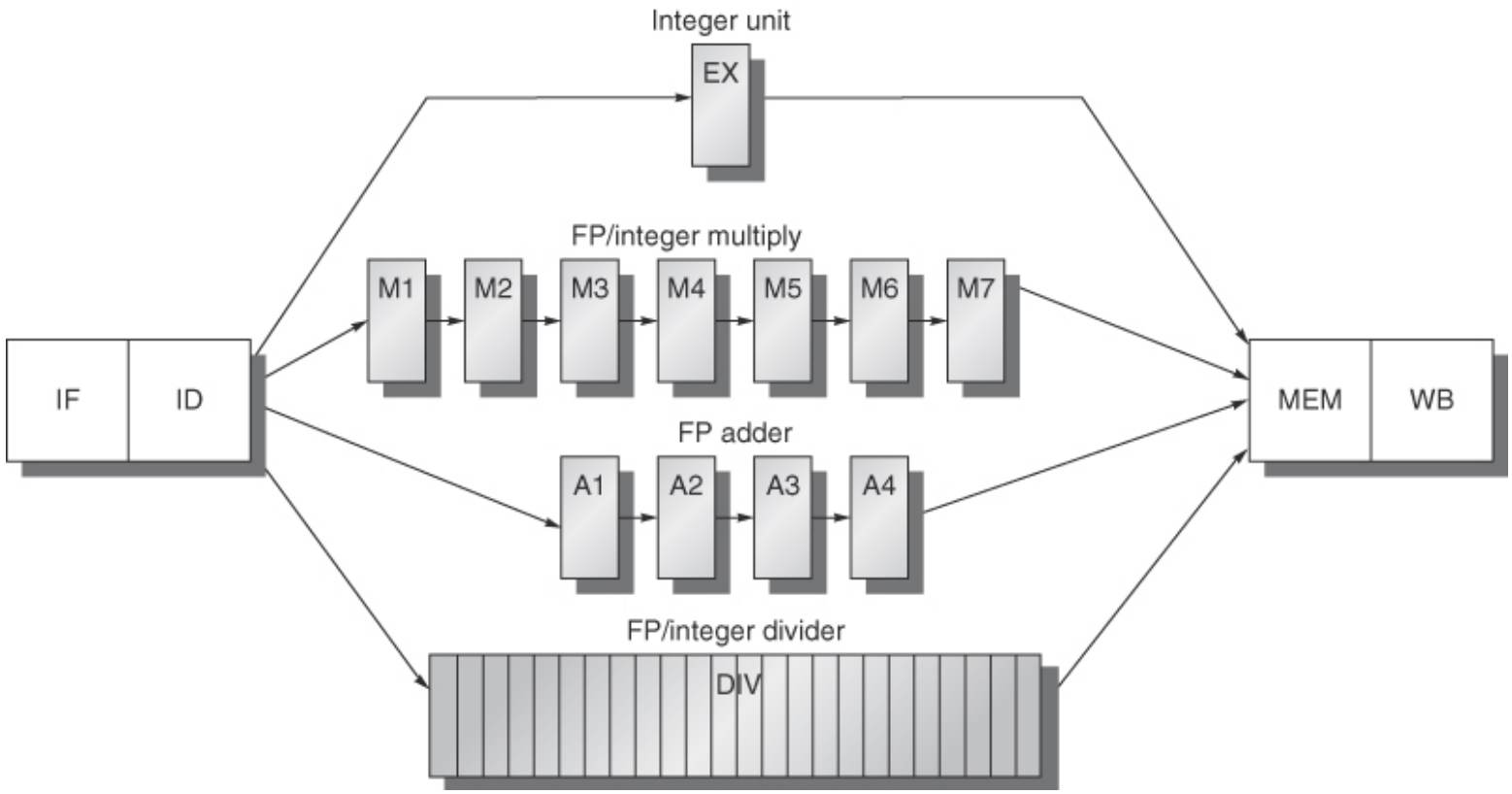
	Stage(s)?	Synchronous?
Arithmetic overflow	EX	yes
Undefined instruction	ID	yes
TLB or page fault	IF, MEM	yes
I/O service request	any	no
Hardware malfunction	any	no

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Dealing with Multicycle Operations



Dealing with Multicycle Operations



Reminders

- Next few lectures are about advanced topics in unicore processors
- Homework1 will be online this week, research essay topic needs to be determined soon.