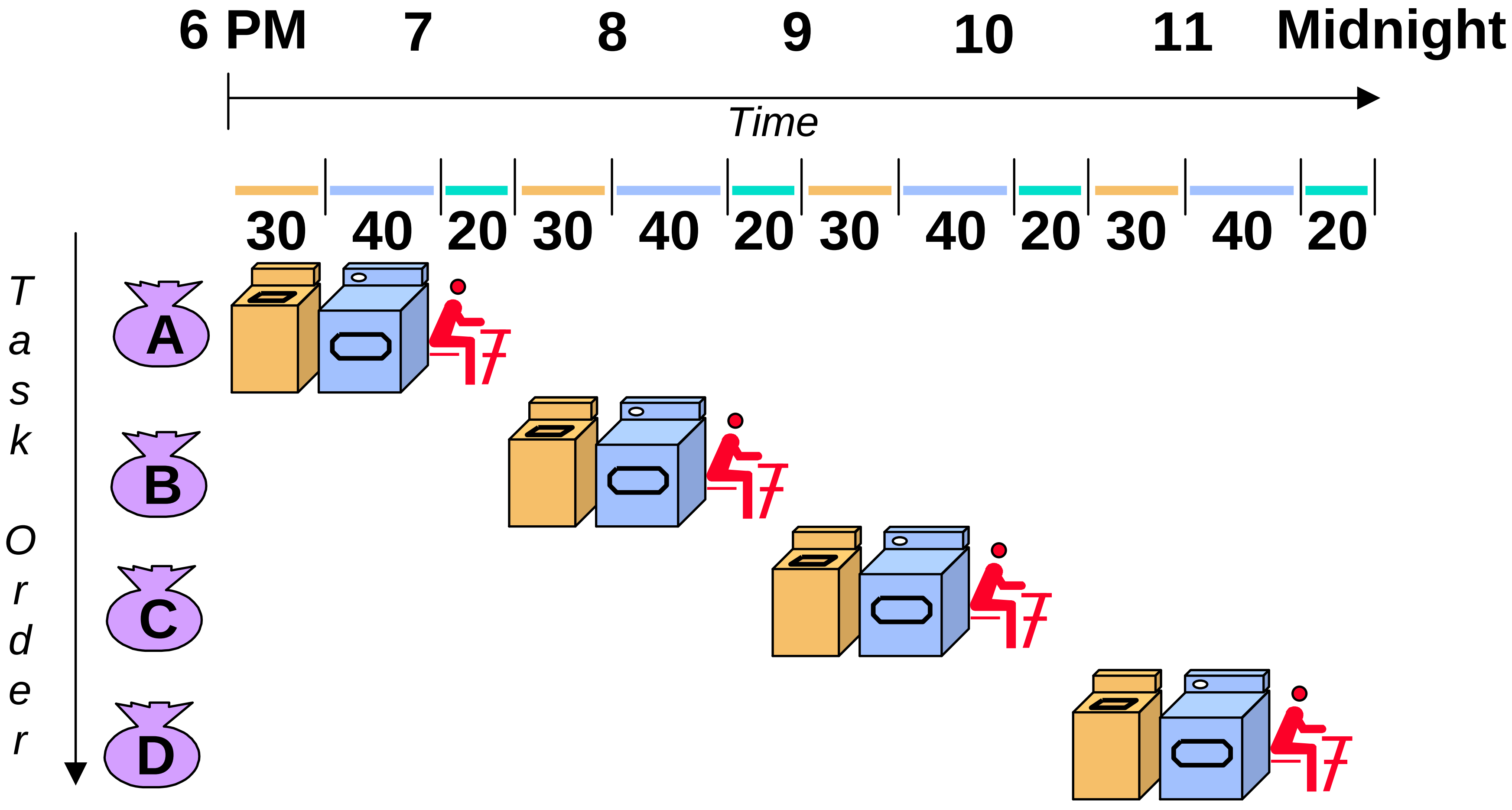
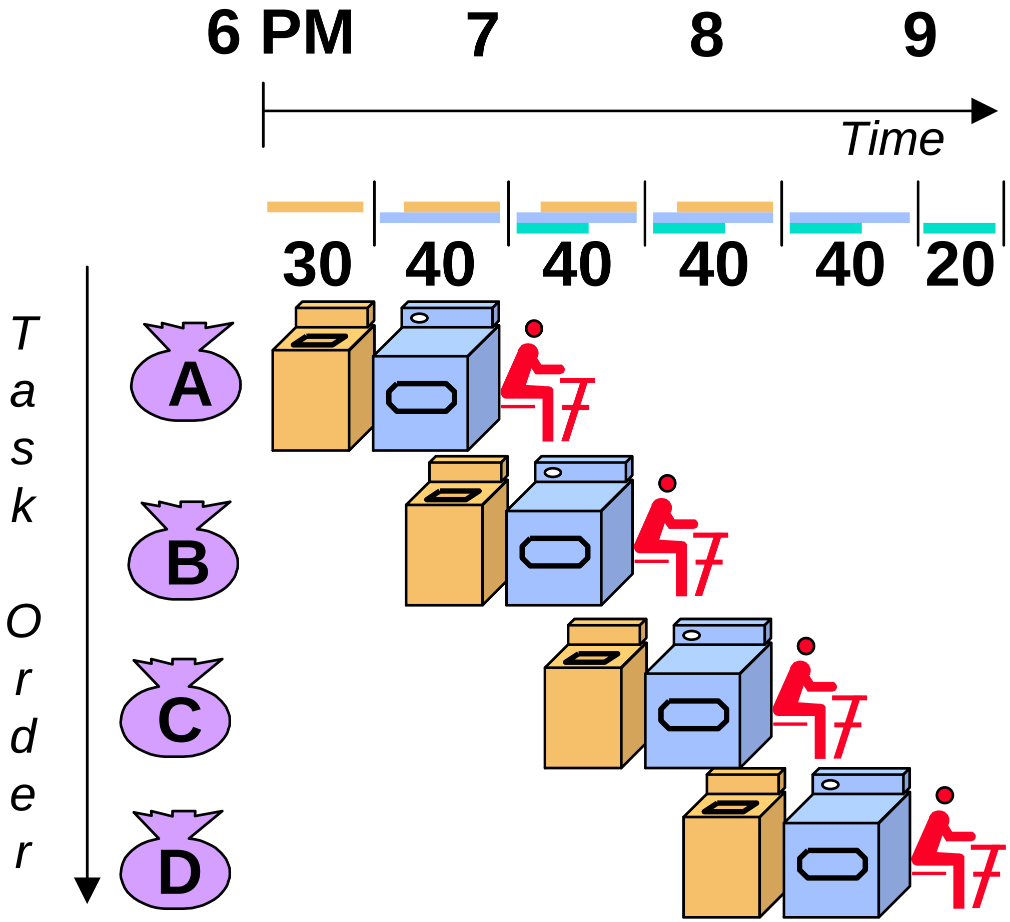
Introduction To Pipelining II

Recap: Sequential Laundry



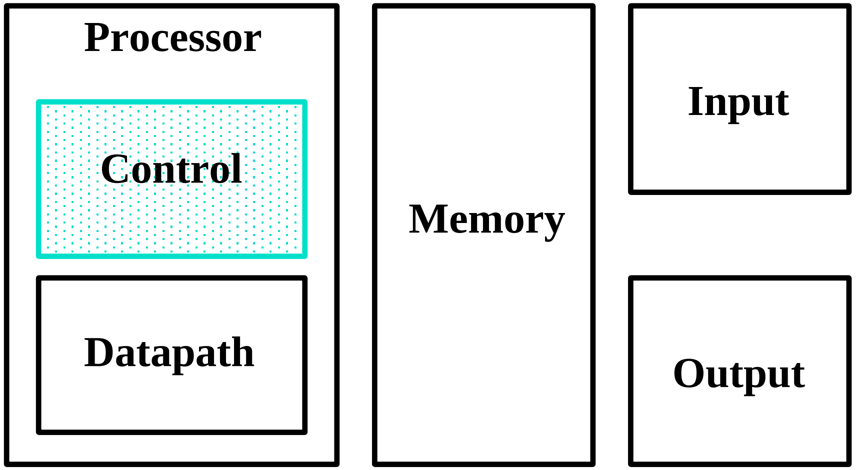
- **Sequential laundry takes 6 hours for 4 loads**
- **If they learned pipelining, how long would laundry take?**

Recap: Pipelining Lessons



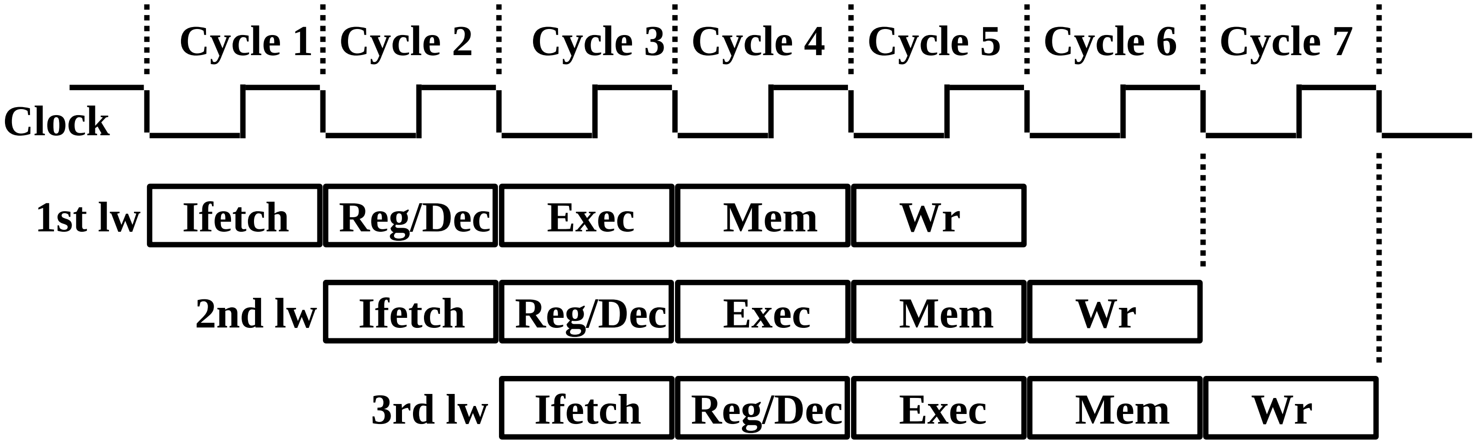
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

The Big Picture: Where are We Now?



- **The Five Classic Components of a Computer**

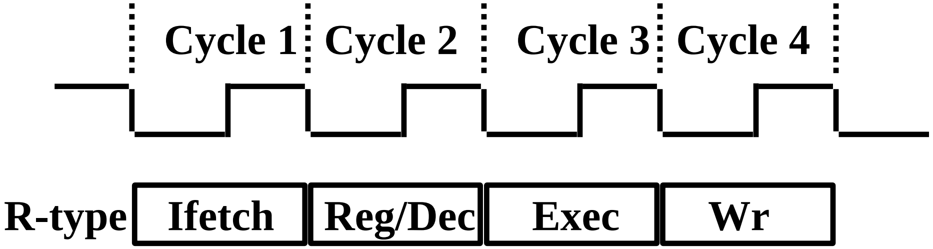
Pipelining the Load Instruction



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
1st lw	Ifetch	Reg/Dec	Exec	Mem	Wr		
2nd lw		Ifetch	Reg/Dec	Exec	Mem	Wr	
3rd lw			Ifetch	Reg/Dec	Exec	Mem	Wr

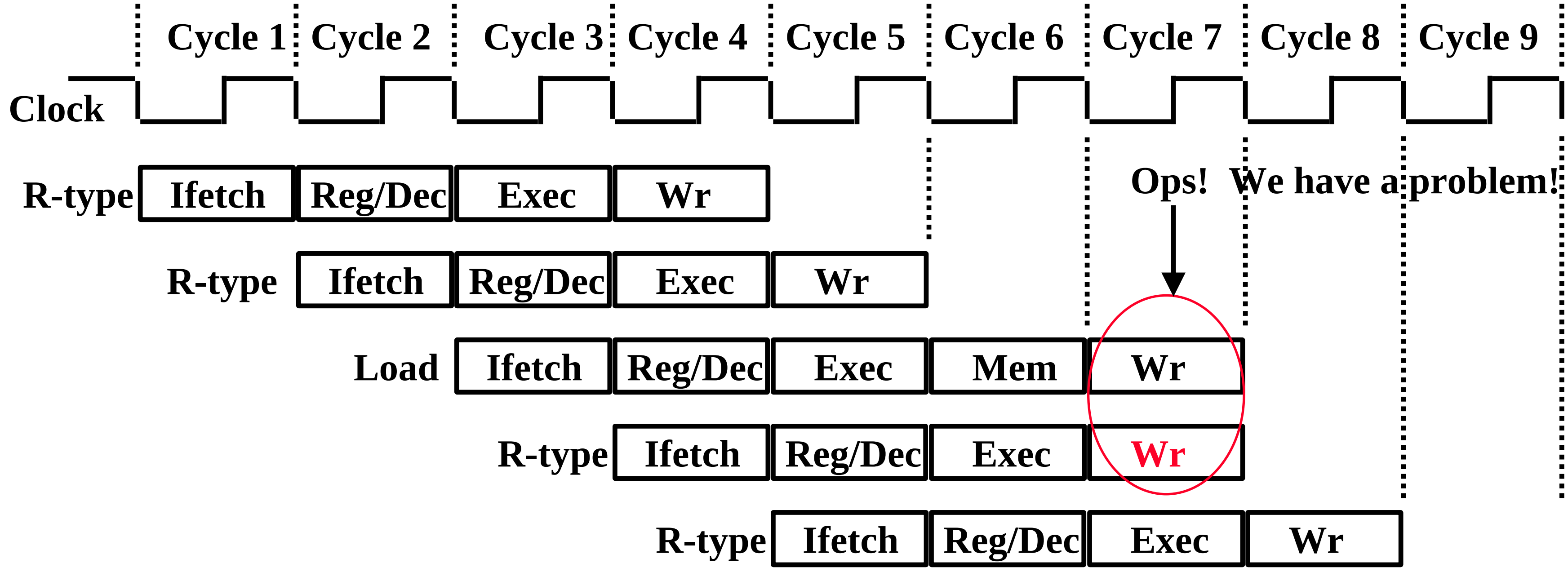
- **The five independent functional units in the pipeline datapath are:**
  - – Instruction Memory for the Ifetch stage
  - – Register File’s Read ports (bus A and busB) for the Reg/Dec stage
  - – ALU for the Exec stage
  - – Data Memory for the Mem stage
  - – Register File’s Write port (bus W) for the Wr stage

The Four Stages of R-type



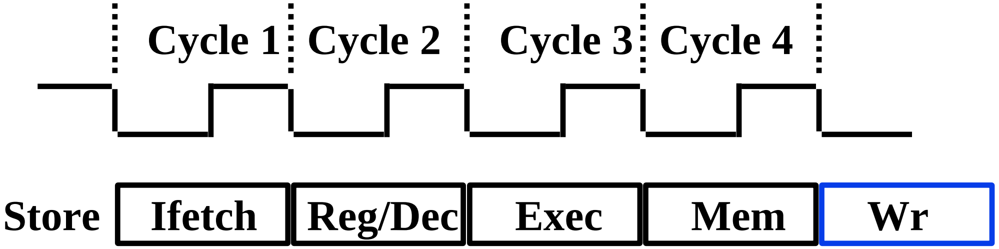
- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
  - ALU operates on the two register operands
  - Update PC
- **Wr: Write the ALU output back to the register file**

Pipelining the R-type and Load Instruction



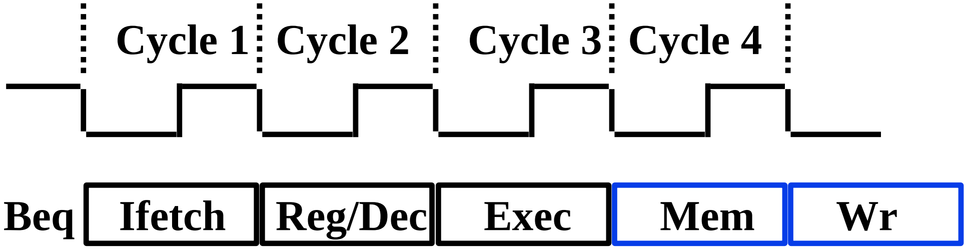
- **We have pipeline conflict or structural hazard:**
  - Two instructions try to write to the register file at the same time!
  - Only one write port
- **Solution – always use all 5 stages of the pipeline!**

The Four Stages of Store



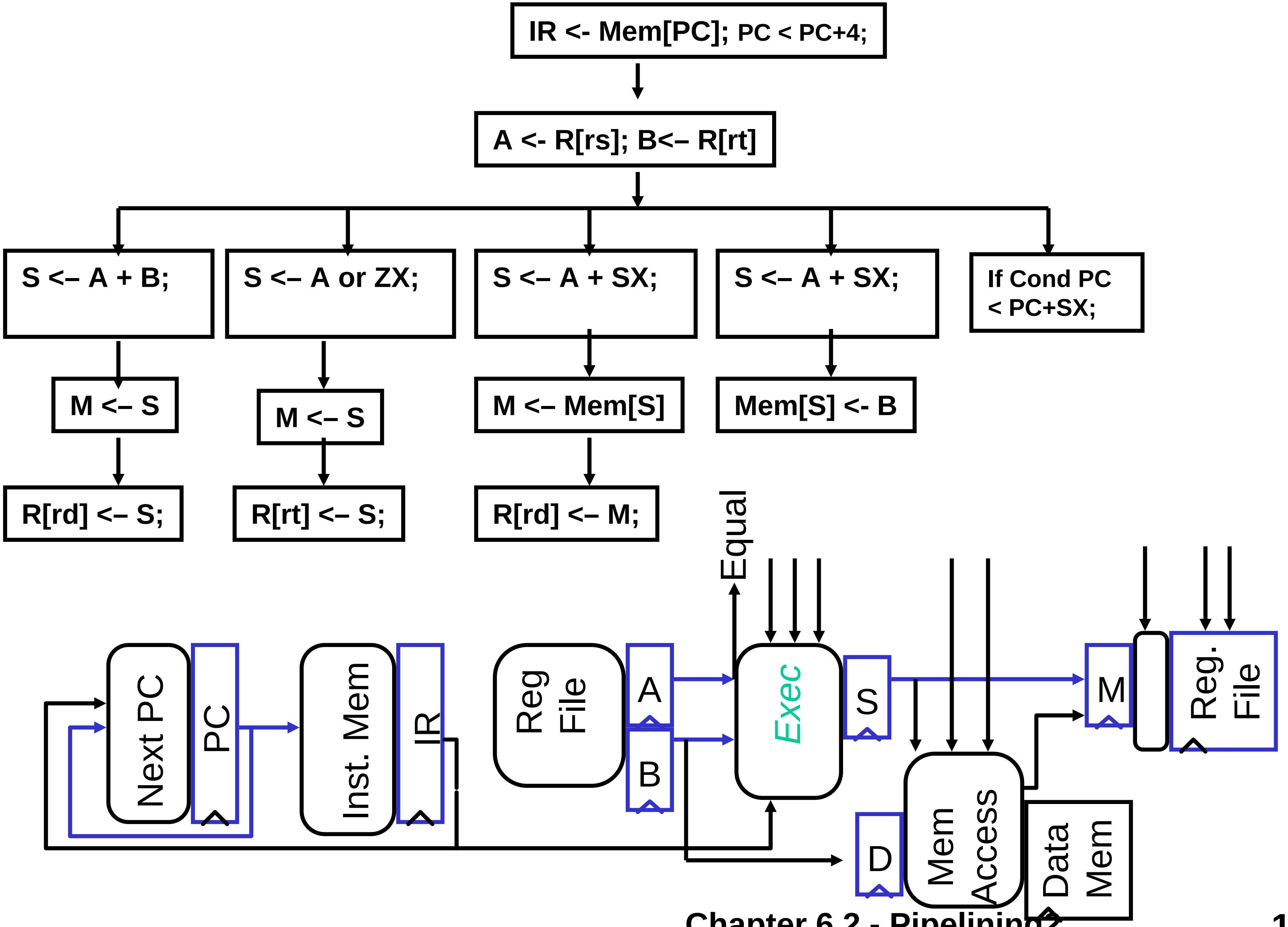
- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Write the data into the Data Memory**

The Three Stages of Beq



- **Ifetch: Instruction Fetch**
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec:**
  - Registers Fetch and Instruction Decode
- **Exec:**
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Recap: Control Diagram



IR <- Mem[PC]; PC < PC+4;

A <- R[rs]; B<– R[rt]

S <– A + B;

M <– S

R[rd] <– S;

S <– A or ZX;

M <– S

R[rt] <– S;

S <– A + SX;

M <– Mem[S]

R[rd] <– M;

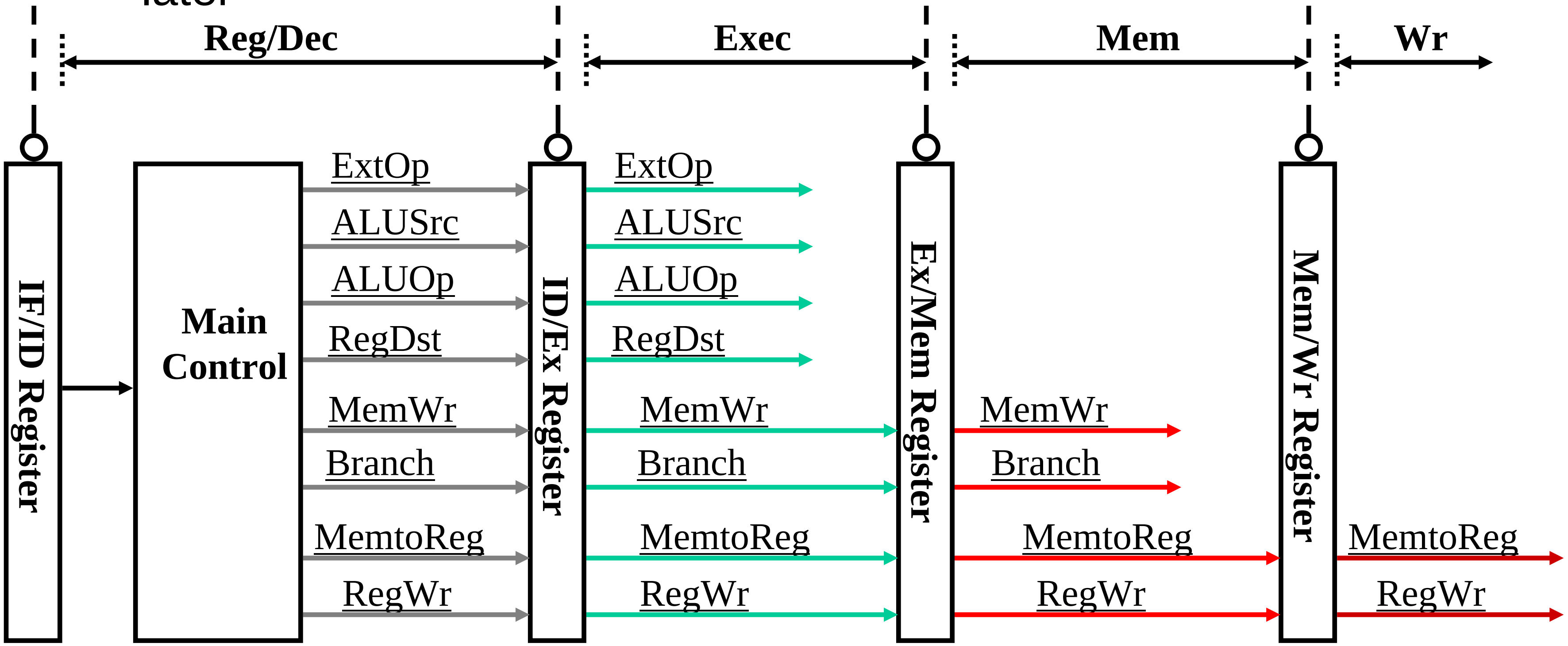
S <– A + SX;

Mem[S] <- B

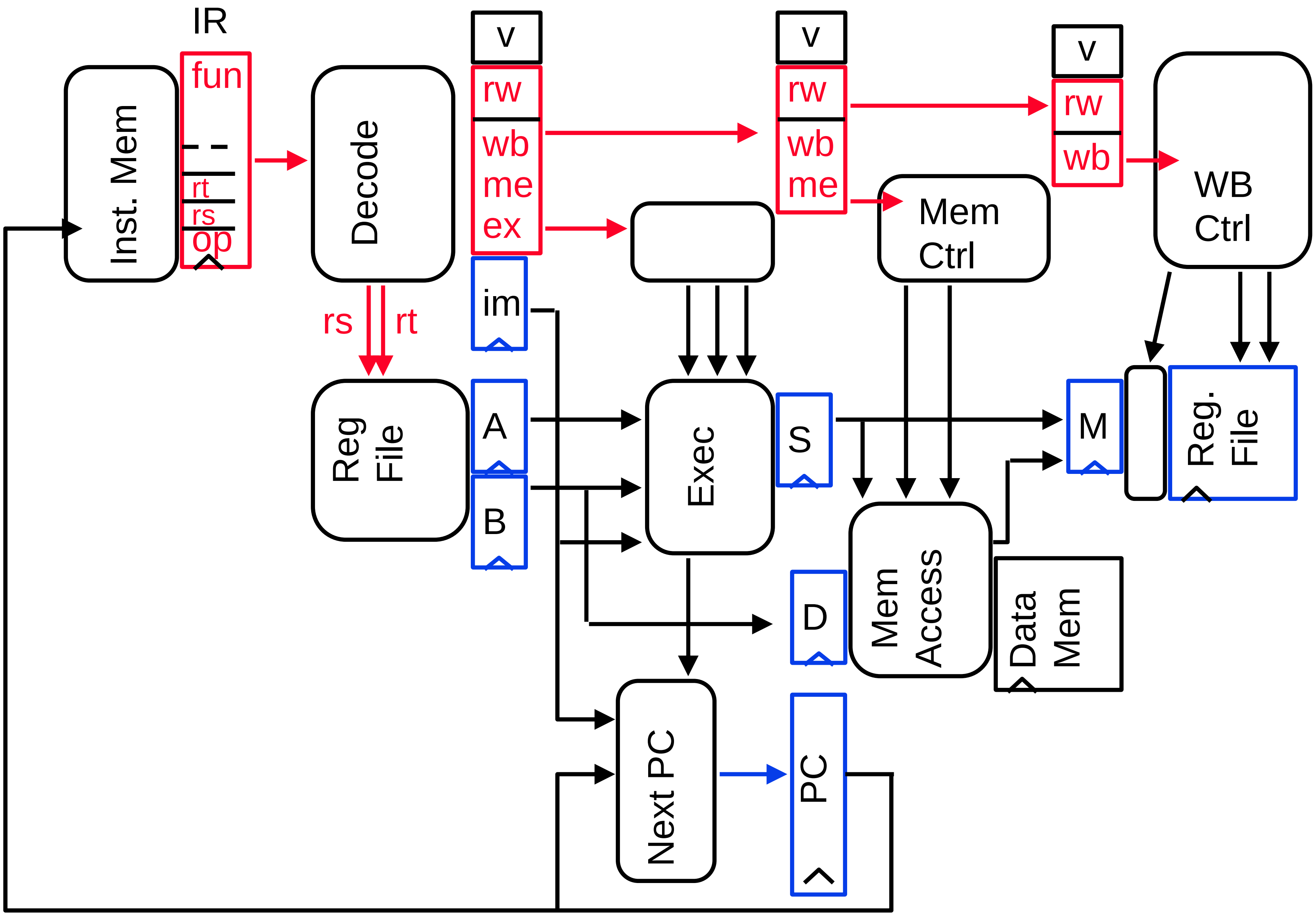
If Cond PC < PC+SX;

But recall use of “Data Stationary Control”

- **The Main Control generates the control signals during Reg/Dec**
  - Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
  - Control signals for Mem (MemWr Branch) are used 2 cycles later
  - Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



Datapath + Data Stationary Control

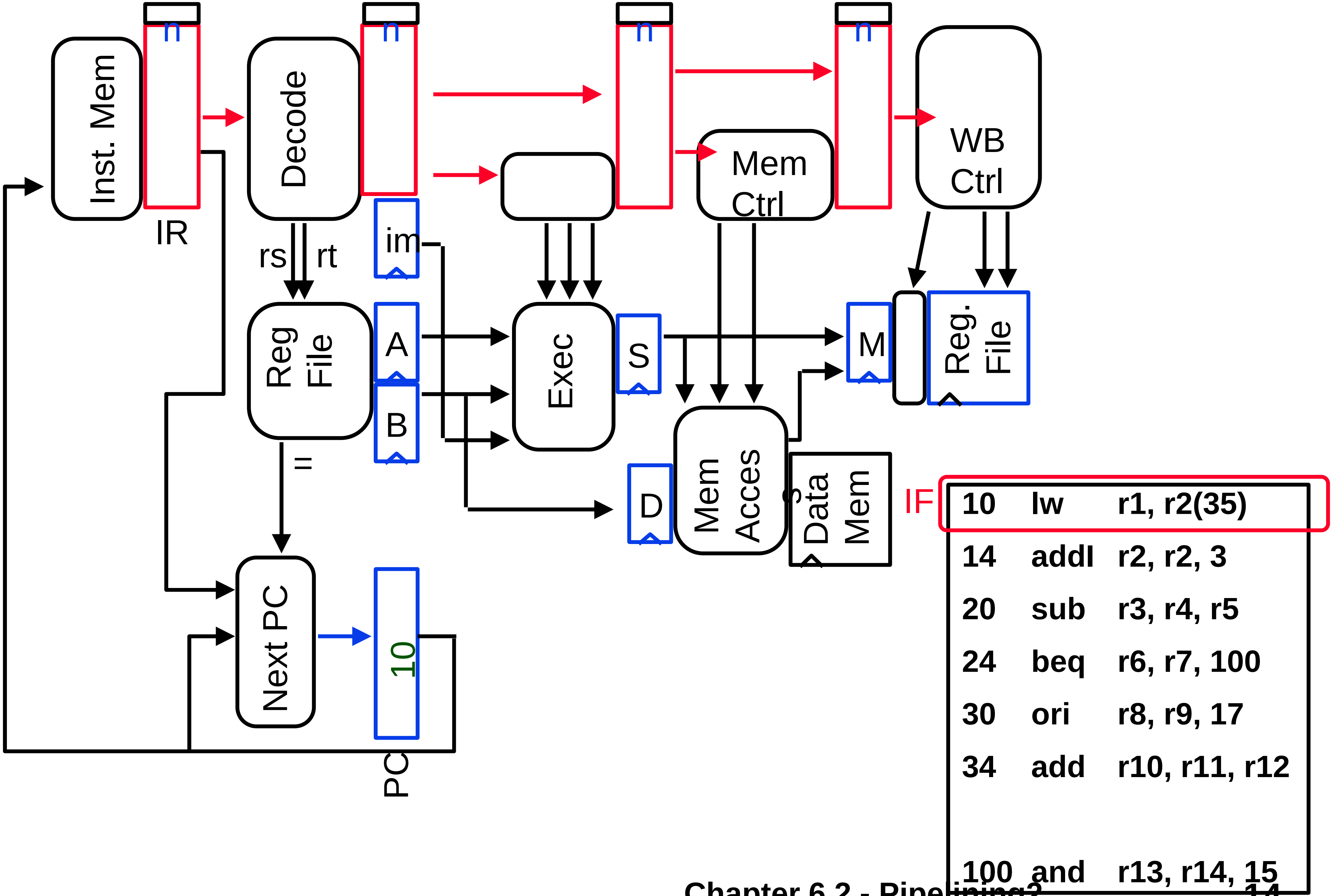


Let’s Try it Out

```
10      lw      r1, r2(35)
14      addI    r2, r2, 3
20      sub     r3, r4, r5
24      beq     r6, r7, 100
30      ori     r8, r9, 17
34      add     r10, r11, r12

100     and     r13, r14, 15
```

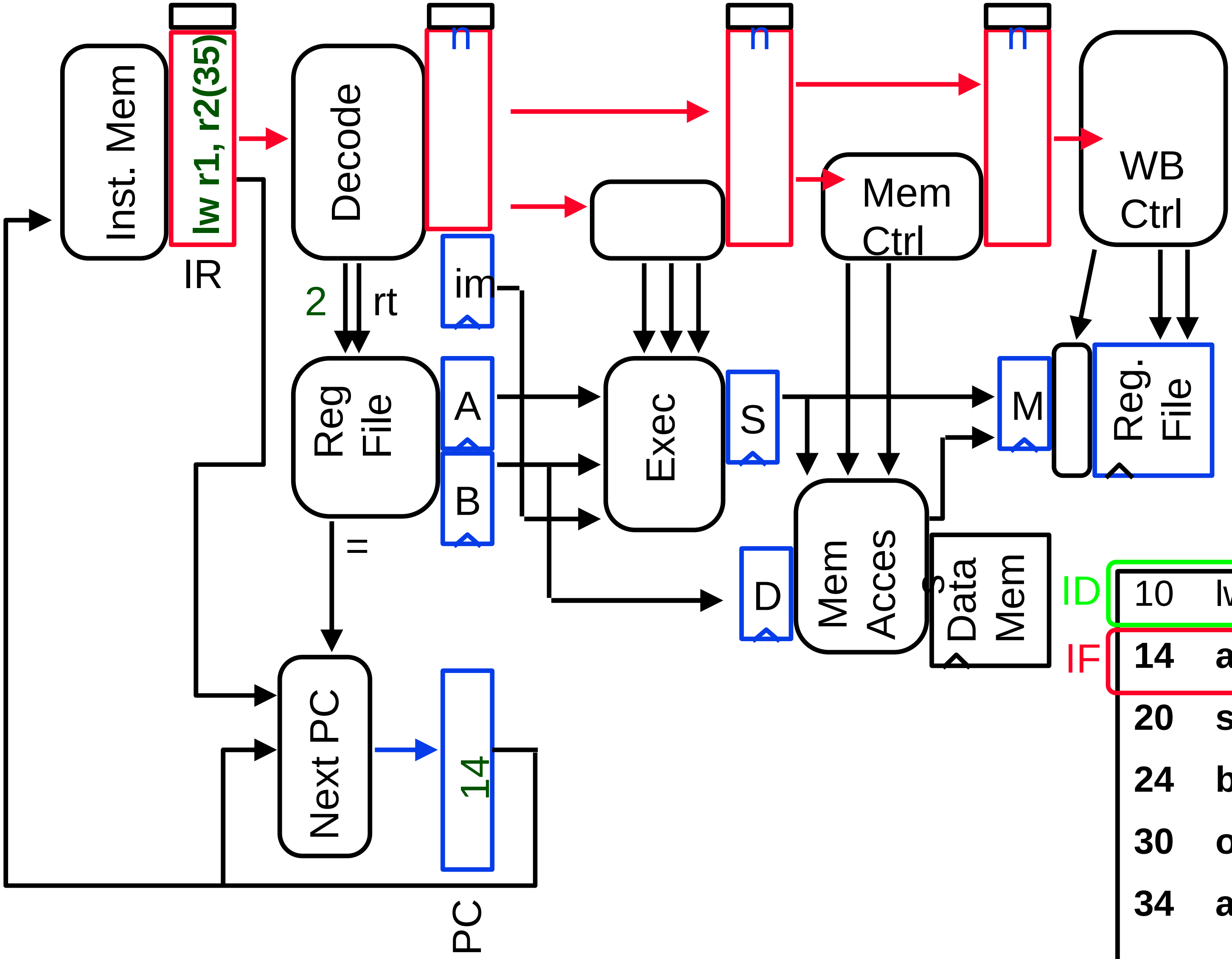
these addresses are octal

Start: Fetch 10



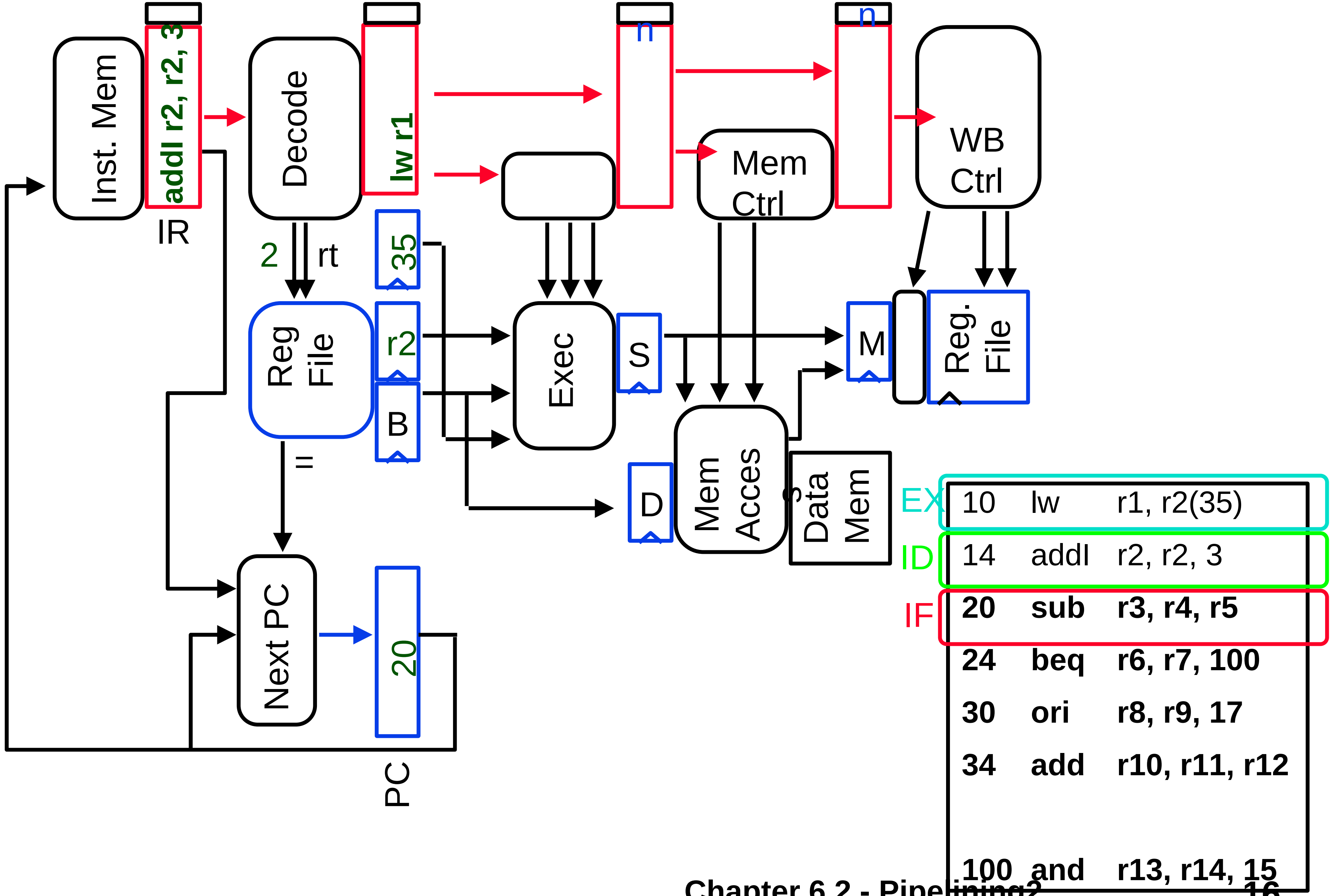
```
10   lw    r1, r2(35)
14   addI  r2, r2, 3
20   sub   r3, r4, r5
24   beq   r6, r7, 100
30   ori   r8, r9, 17
34   add   r10, r11, r12

100  and   r13, r14, 15
```

Fetch 14, Decode 10



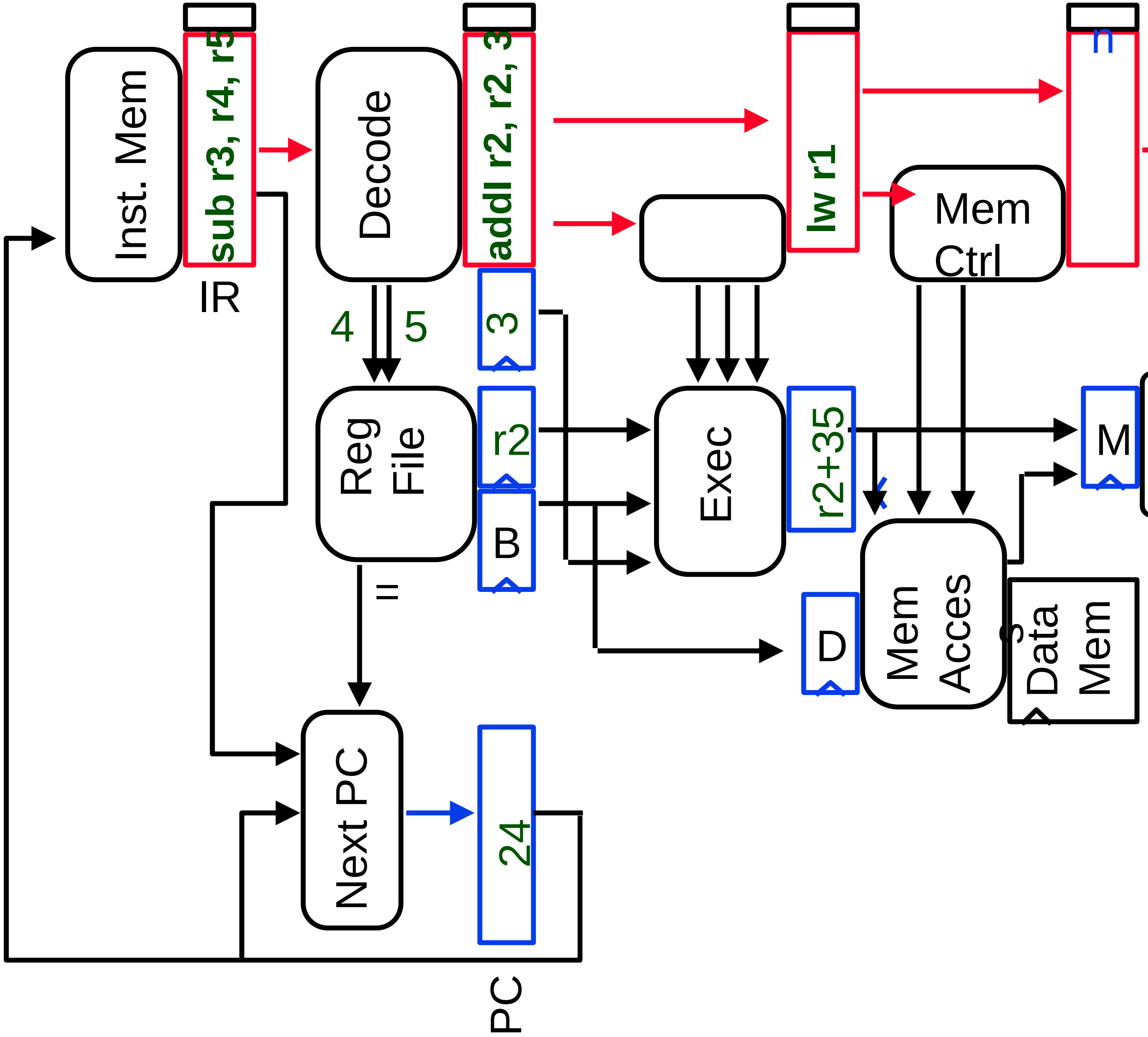
ID	10	lw	r1, r2(35)
IF	**14**	**addI**	**r2, r2, 3**
	20	**sub**	**r3, r4, r5**
	24	**beq**	**r6, r7, 100**
	30	**ori**	**r8, r9, 17**
	34	**add**	**r10, r11, r12**
	100	**and**	**r13, r14, 15**

Fetch 20, Decode 14, Exec 10



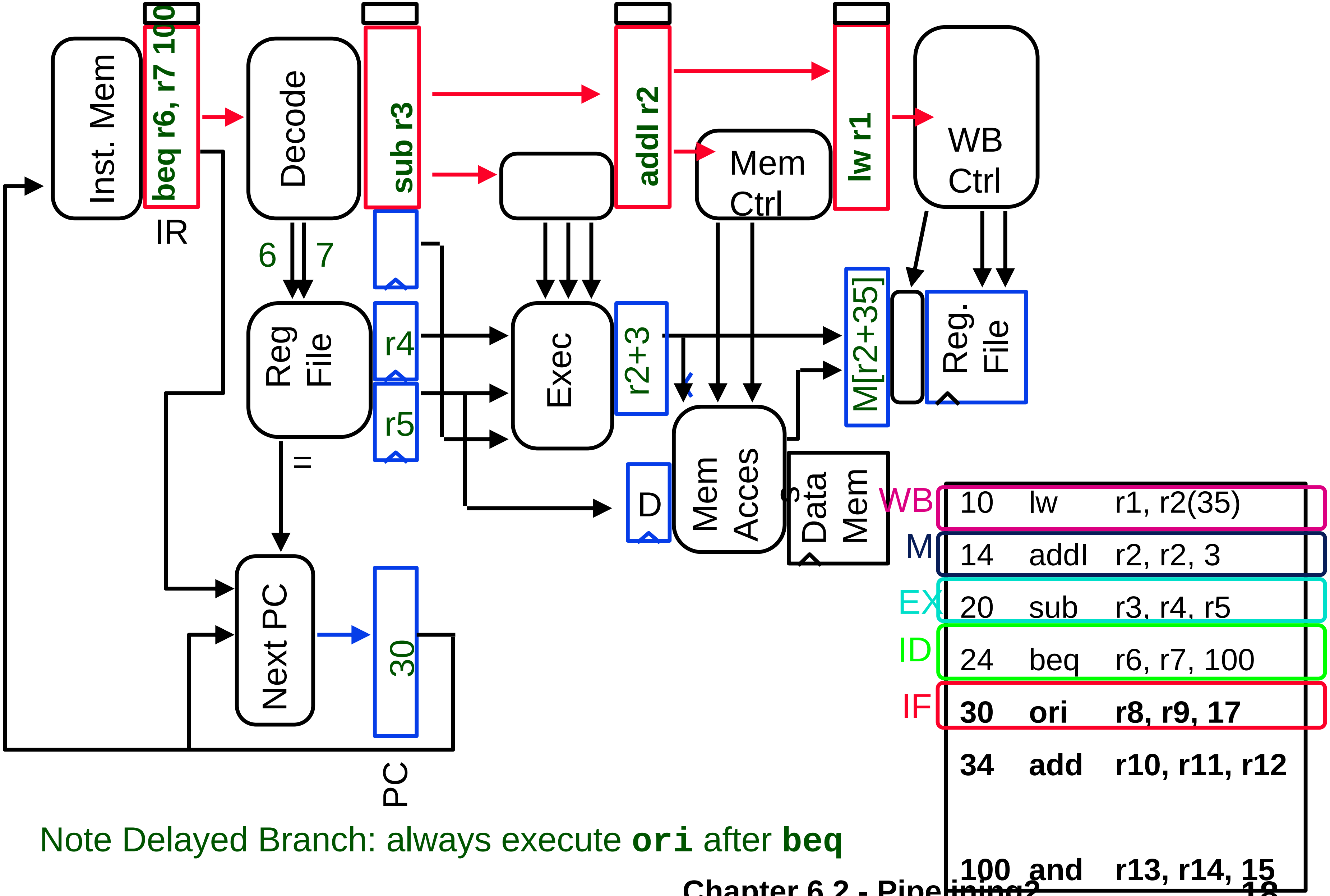
```
EX  10    lw     r1, r2(35)
ID  14    addI   r2, r2, 3
IF  20    sub    r3, r4, r5
    24    beq    r6, r7, 100
    30    ori    r8, r9, 17
    34    add    r10, r11, r12


    100   and    r13, r14, 15
```

Fetch 24, Decode 20, Exec 14, Mem 10



M	10	lw	r1, r2(35)
EX	14	addI	r2, r2, 3
ID	20	sub	r3, r4, r5
IF	**24**	**beq**	**r6, r7, 100**
	30	**ori**	**r8, r9, 17**
	34	**add**	**r10, r11, r12**
	100	**and**	**r13, r14, 15**

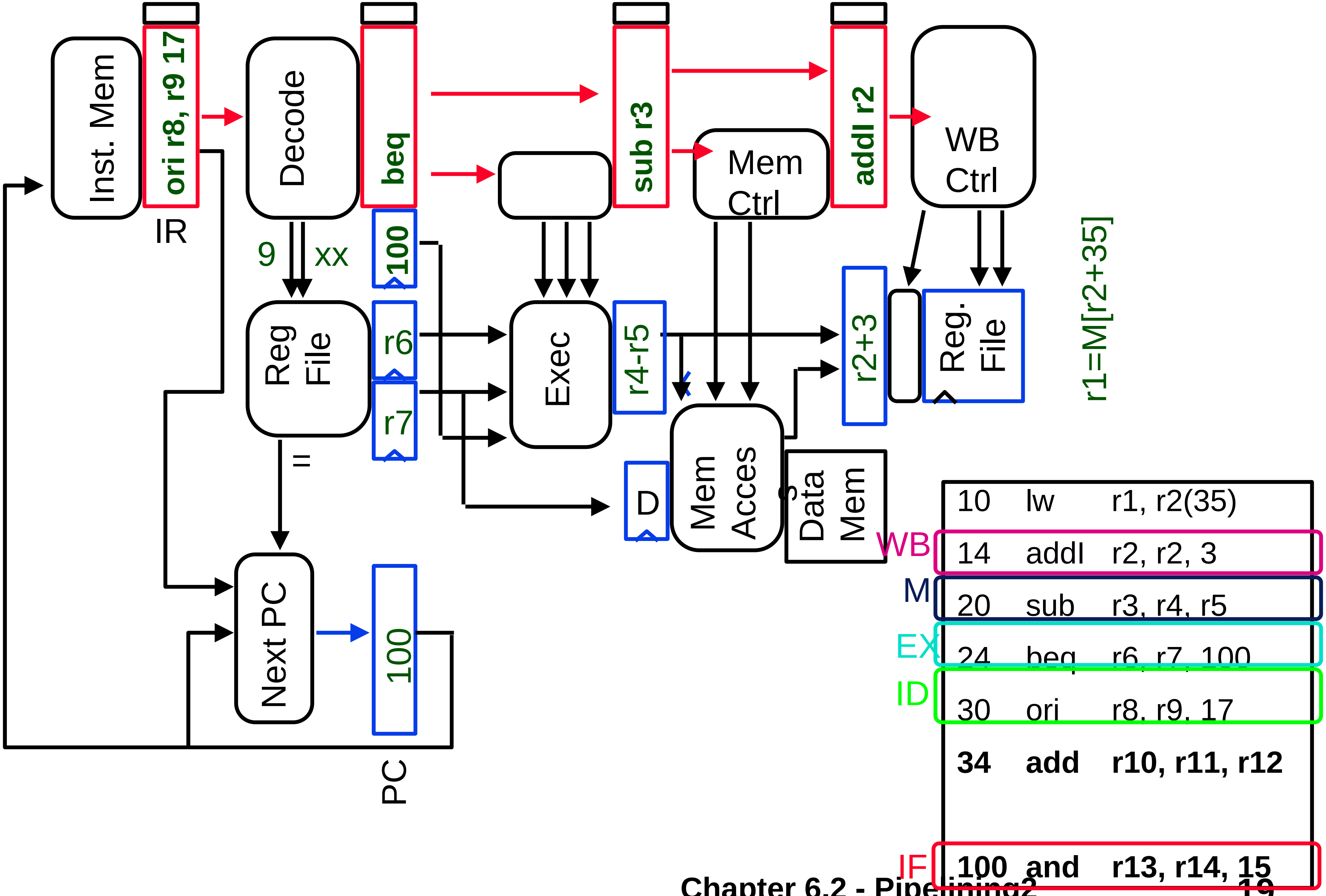
Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10



```
WB  10   lw    r1, r2(35)
M   14   addI  r2, r2, 3
EX  20   sub   r3, r4, r5
ID  24   beq   r6, r7, 100
IF  30   ori   r8, r9, 17
    34   add   r10, r11, r12

    100  and   r13, r14, 15
```

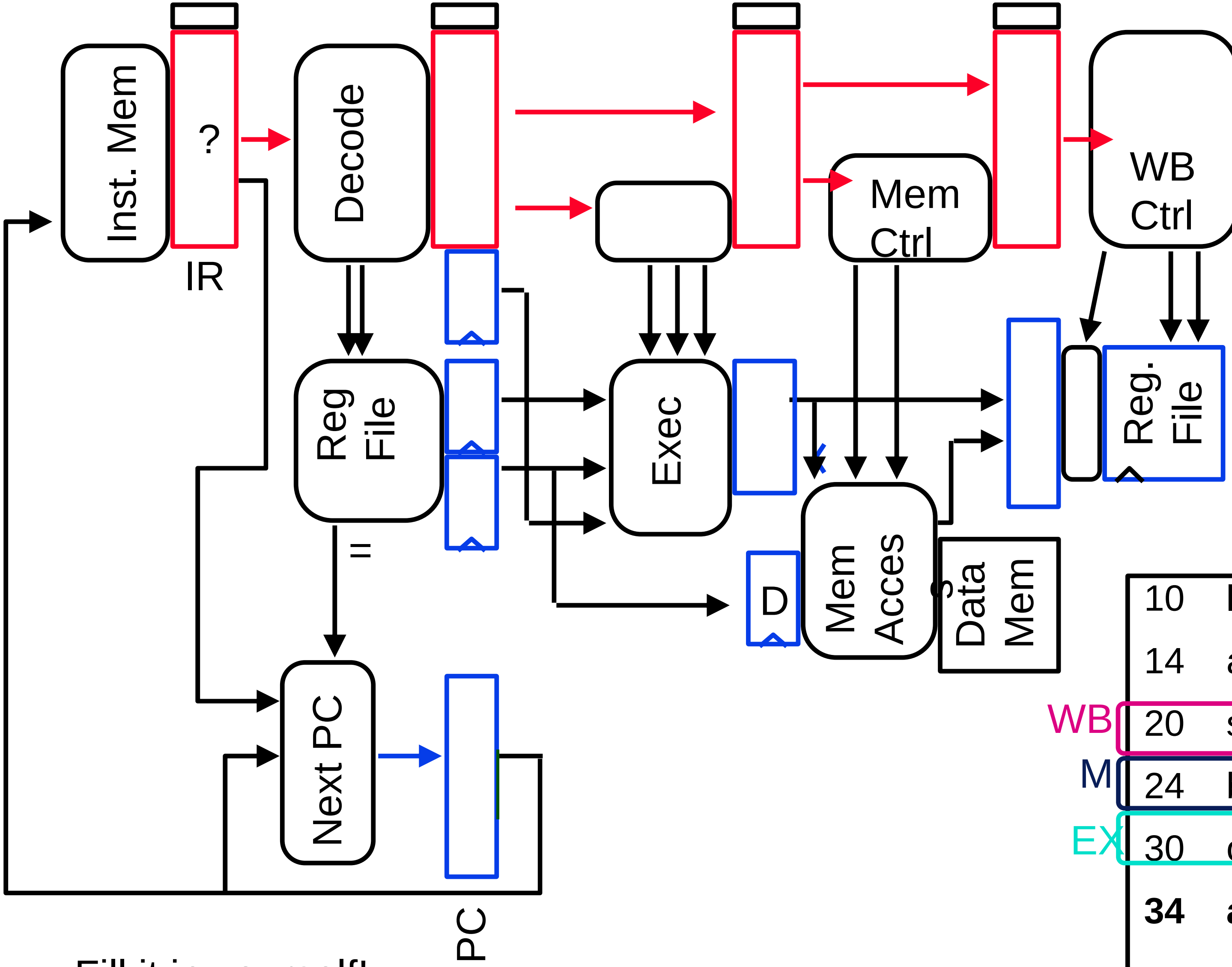
Note Delayed Branch: always execute **`ori`** after **`beq`**

Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14



```
   10   lw    r1, r2(35)
WB 14   addI  r2, r2, 3
M  20   sub   r3, r4, r5
EX 24   beq   r6, r7, 100
ID 30   ori   r8, r9, 17
   34   add   r10, r11, r12

IF 100  and   r13, r14, 15
```

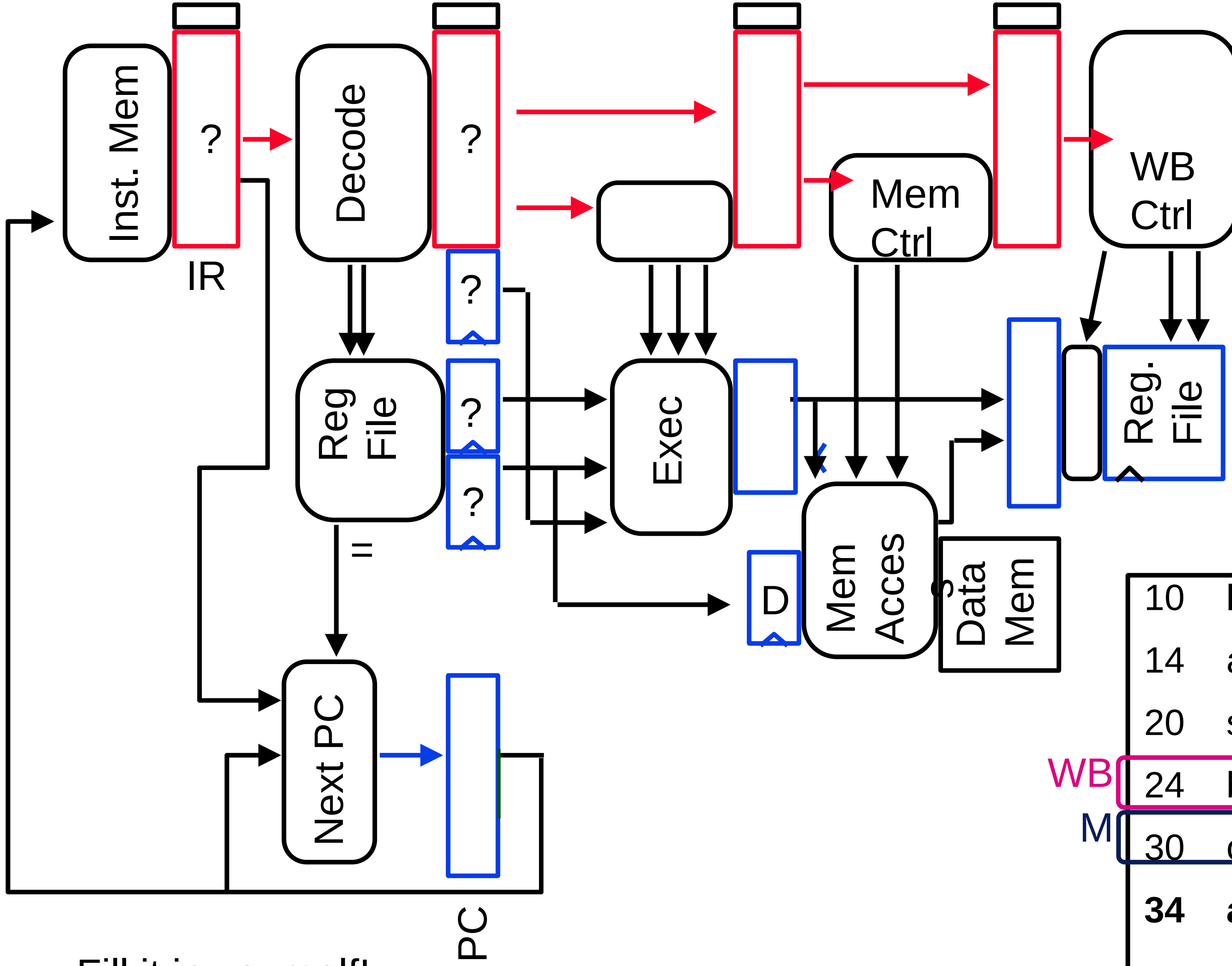
Fetch 104, Dcd 100, Ex 30, Mem 24, WB 20



Fill it in yourself!

	10	lw	r1, r2(35)
	14	addI	r2, r2, 3
WB	20	sub	r3, r4, r5
M	24	beq	r6, r7, 100
EX	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
ID	**100**	**and**	**r13, r14, 15**

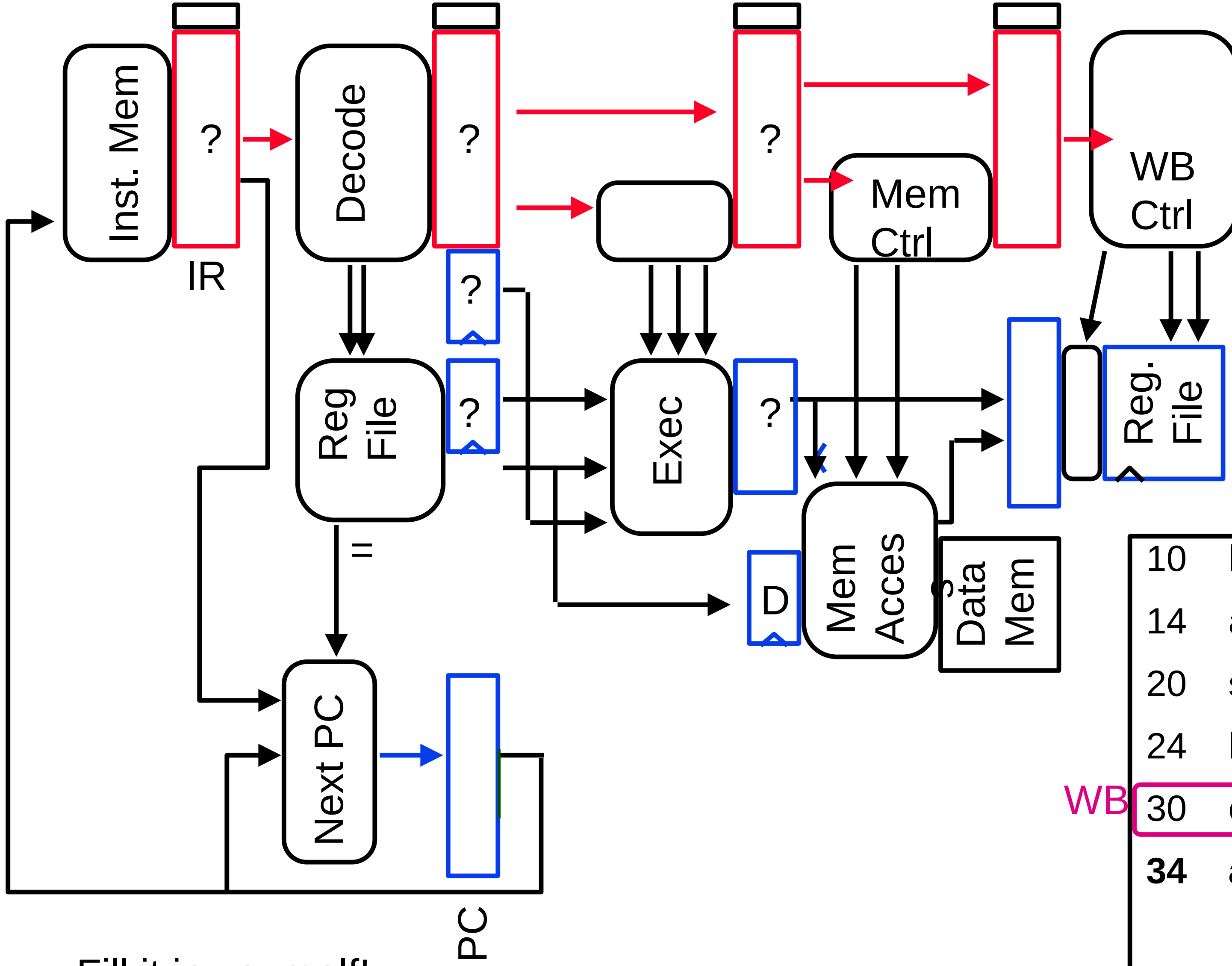
Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24



Fill it in yourself!

	10	lw	r1, r2(35)
	14	addI	r2, r2, 3
	20	sub	r3, r4, r5
WB	24	beq	r6, r7, 100
M	30	ori	r8, r9, 17
	34	**add**	**r10, r11, r12**
EX	**100**	**and**	**r13, r14, 15**

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 30

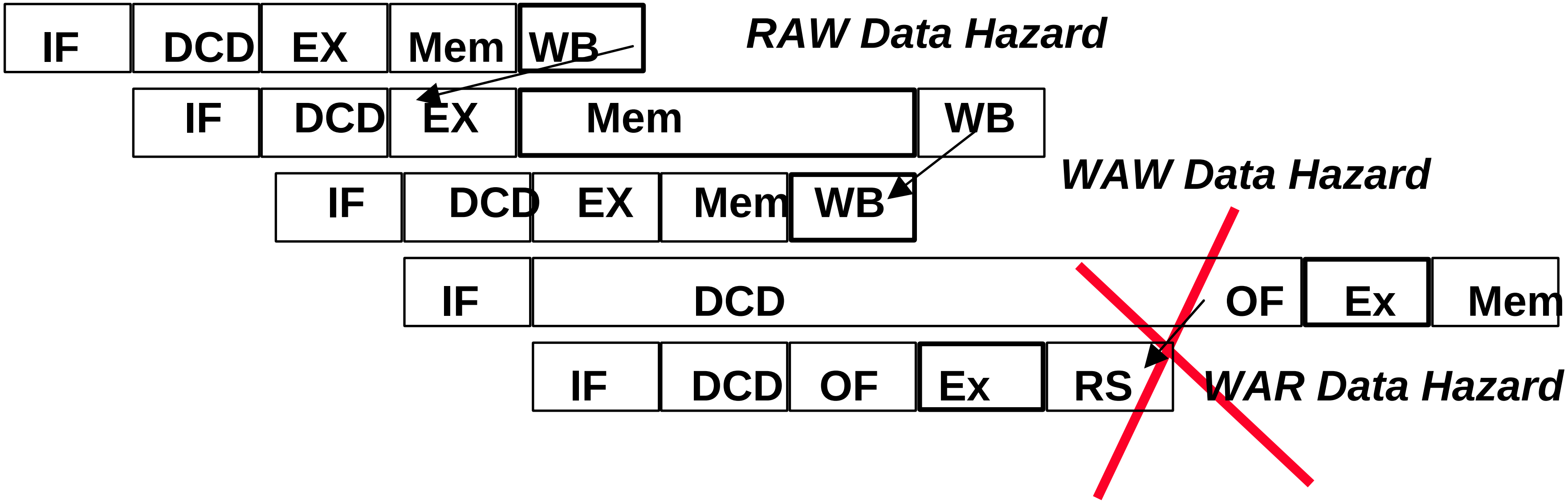


```
   10  lw    r1, r2(35)
   14  addI  r2, r2, 3
   20  sub   r3, r4, r5
   24  beq   r6, r7, 100
WB 30  ori   r8, r9, 17
   34  add   r10, r11, r12

   100 and   r13, r14, 15
M
```

Fill it in yourself!

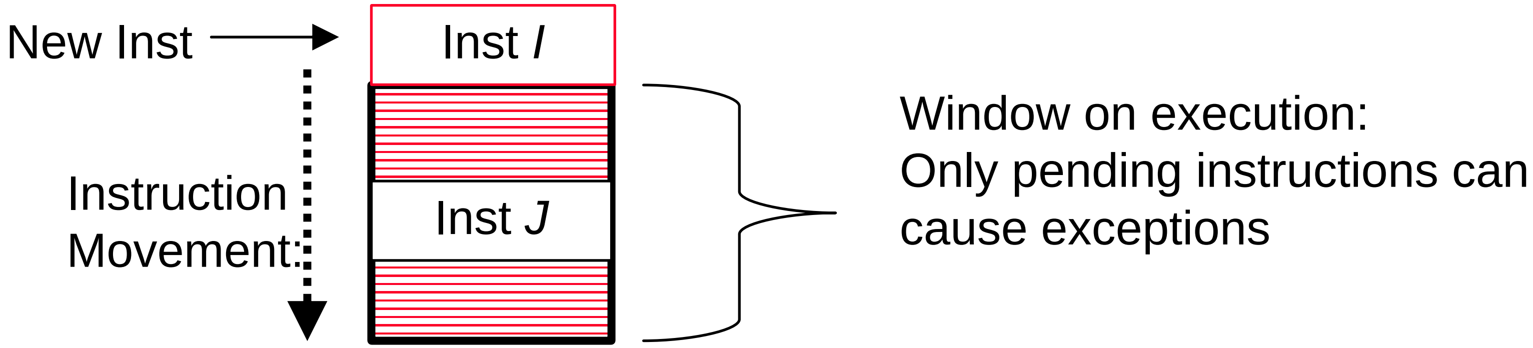
Data Hazards Handling

- ***Avoid*** **some "by design":**
  - eliminate WAR by always fetching operands early (decode) in pipe
  - eliminate WAW by doing all WBs in order (last stage, static).
- ***Detect*** **and** ***resolve*** **remaining ones**
  - stall the pipeline,
  - or, forward (if possible).



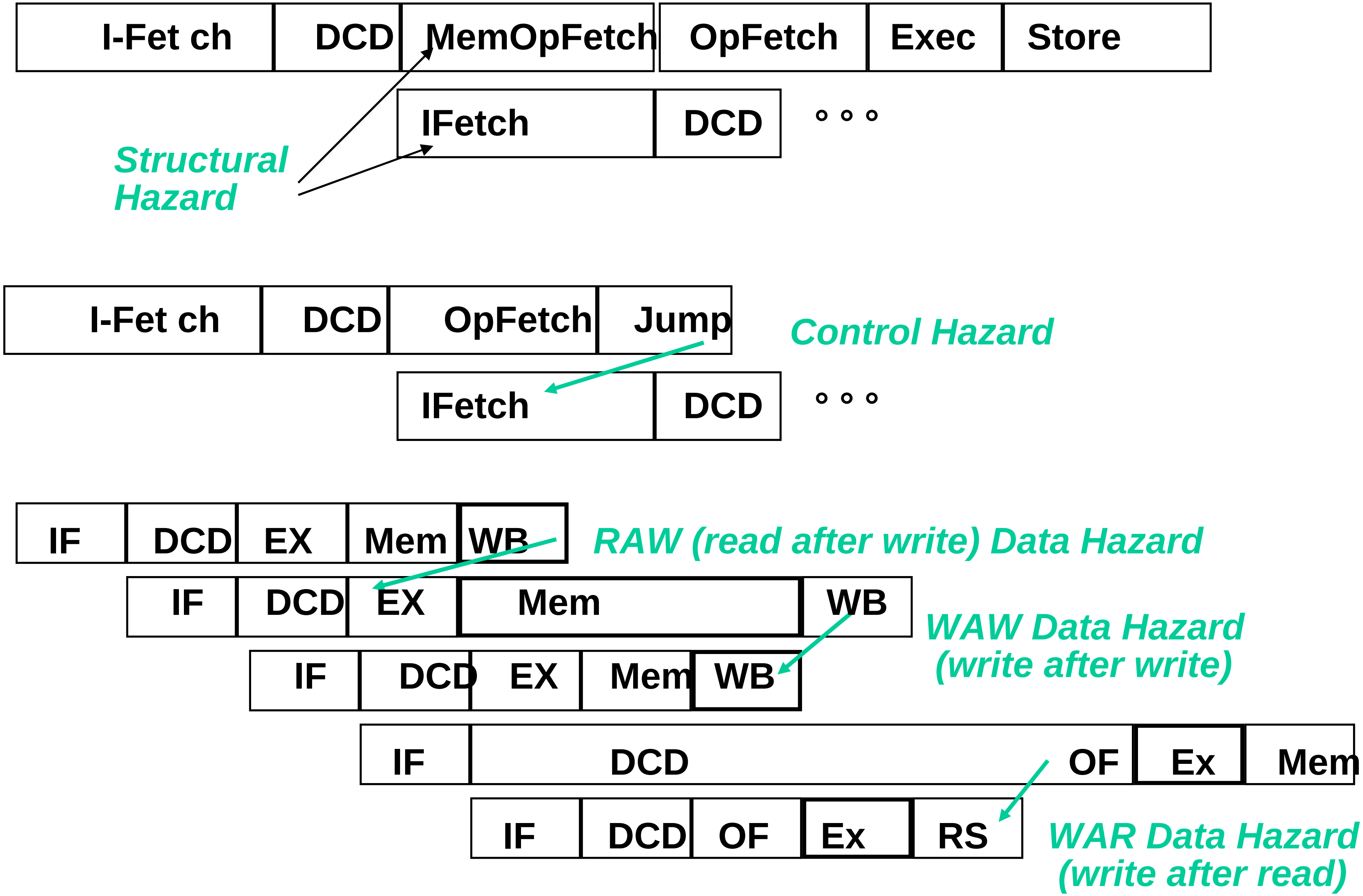
Hazard Detection

- **Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.**



- **A RAW hazard exists on register *r* if $r \in R_{regs}(i) \cap W_{regs}(j)$**
  - Keep a record of pending writes (for instructions in the pipe) and compare with operand registers of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.
- **A WAW hazard exists on register *r* if $r \in W_{regs}(i) \cap W_{regs}(j)$**
- **A WAR hazard exists on register *r* if $r \in W_{regs}(i) \cap R_{regs}(j)$**

Pipeline Hazards Again



I-Fet ch | DCD | MemOpFetch | OpFetch | Exec | Store

IFetch | DCD ° ° °

Structural Hazard

I-Fet ch | DCD | OpFetch | Jump

IFetch | DCD ° ° °

Control Hazard

IF | DCD | EX | Mem | WB

IF | DCD | EX | Mem | WB

IF | DCD | EX | Mem | WB

IF | DCD | OF | Ex | Mem

IF | DCD | OF | Ex | RS

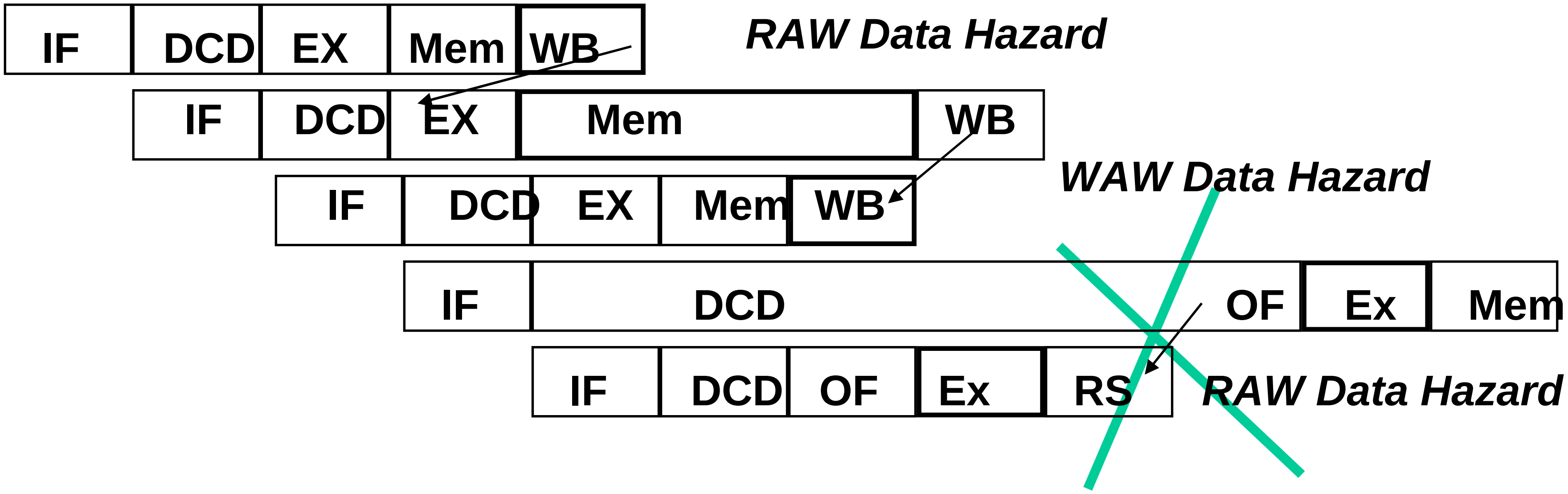
RAW (read after write) Data Hazard

WAW Data Hazard (write after write)

WAR Data Hazard (write after read)

Data Hazards

- **Avoid some “by design”**
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eleminate WAW by doing all WBs in order (last stage, static)
- **Detect and resolve remaining ones**
  - stall or forward (if possible)



IF DCD EX Mem WB

IF DCD EX Mem WB

IF DCD EX Mem WB

IF DCD OF Ex Mem

IF DCD OF Ex RS

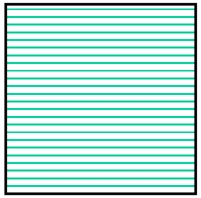
RAW Data Hazard

WAW Data Hazard

RAW Data Hazard

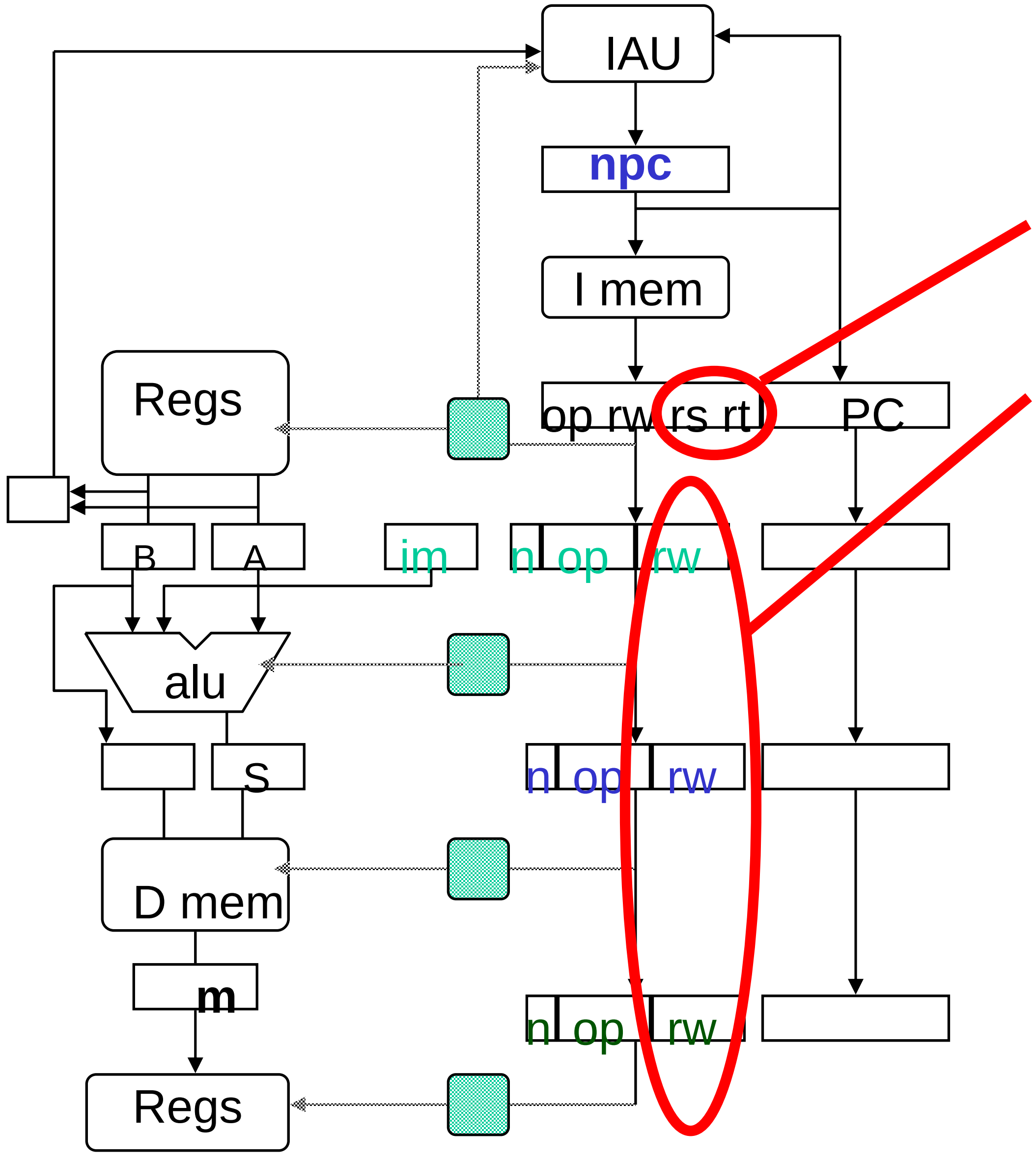
Hazard Detection

- **Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.**
- **A RAW hazard exists on register** ρ **if** $\rho \in$ **Rregs(*i*)** $\cap$ **Wregs(*j*)**
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When on operation completes, remove its write reservation.



- **A WAW hazard exists on register** ρ **if** $\rho \in$ **Wregs(*i*)** $\cap$ **Wregs(*j*)**
- **A WAR hazard exists on register** ρ **if** $\rho \in$ **Wregs(*i*)** $\cap$ **Rregs(*j*)**

Record of Pending Writes



- **Current operand registers**
- **Pending writes**
- **hazard <=**

  $((rs == rw_{ex)} \quad \& \ regW_{ex})$

  OR

  $((rs == rw_{mem)} \ \& \ regW_{me})$

  OR

  $((rs == rw_{wb)} \quad \& \ regW_{wb})$

  OR

  $((rt == rw_{ex)} \quad \& \ regW_{ex})$

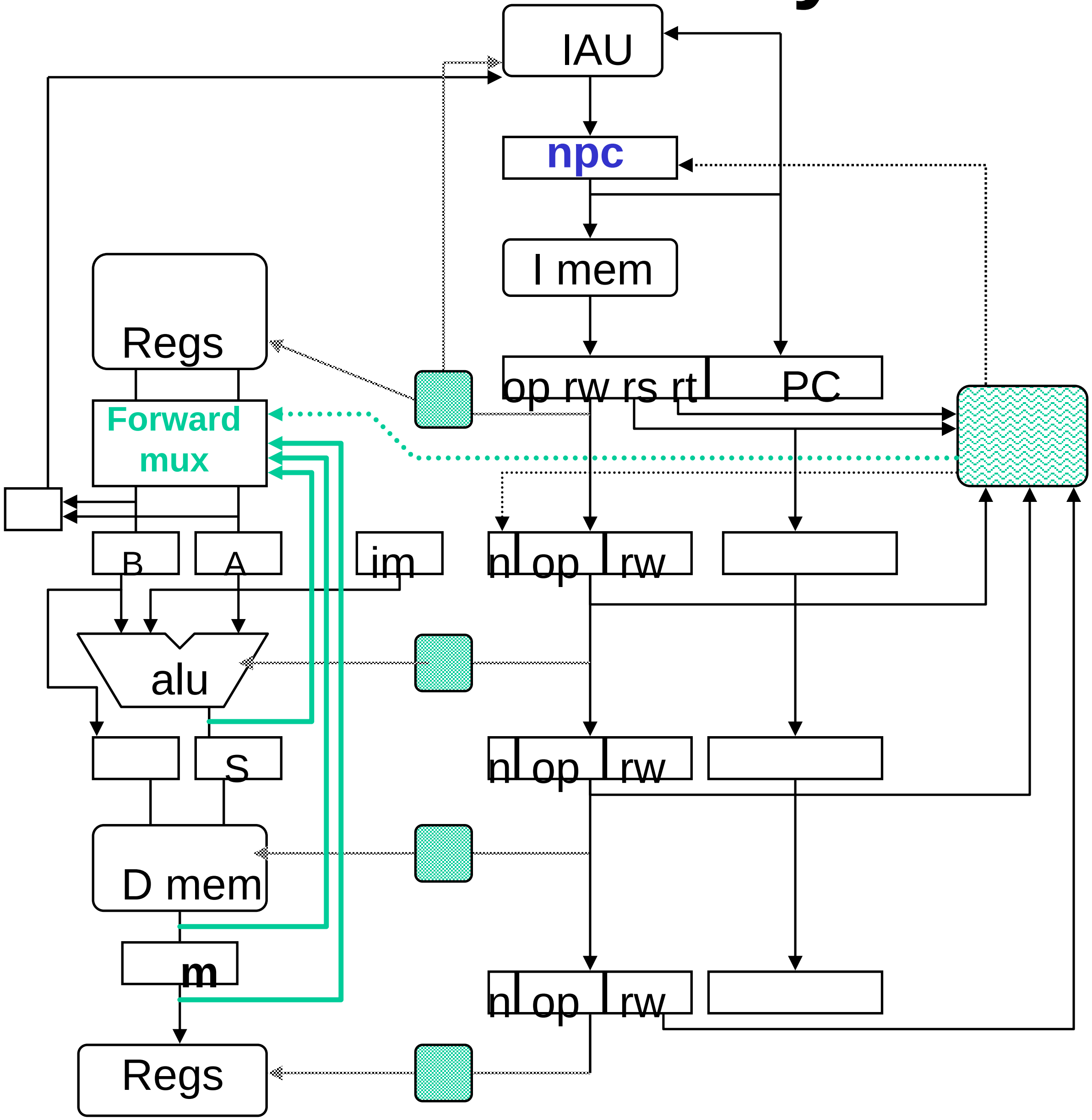
  OR

  $((rt == rw_{mem)} \ \& \ regW_{me})$

  OR

  $((rt == rw_{wb}) \quad \& \ regW_{wb})$

Resolve RAW by forwarding



- Detect nearest valid write op operand register and forward into op latches, bypassing remainder of the pipe
- Increase muxes to add paths from pipeline registers
- Data Forwarding = Data Bypassing

What about memory operations?

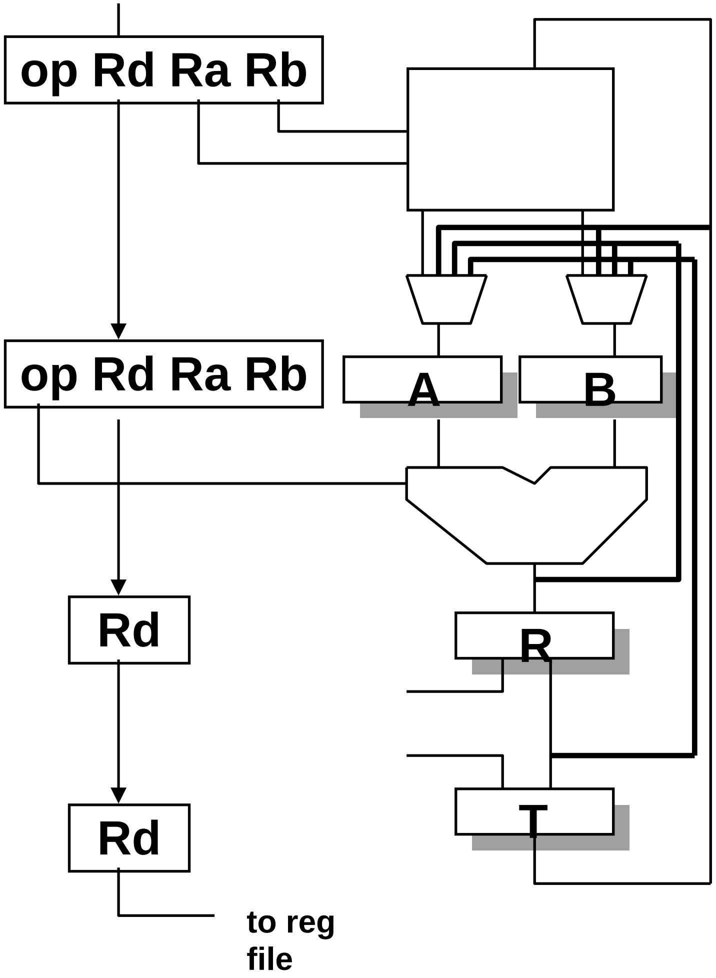
- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!

- What does delaying WB on arithmetic operations cost?
  - – cycles ?
  - – hardware ?

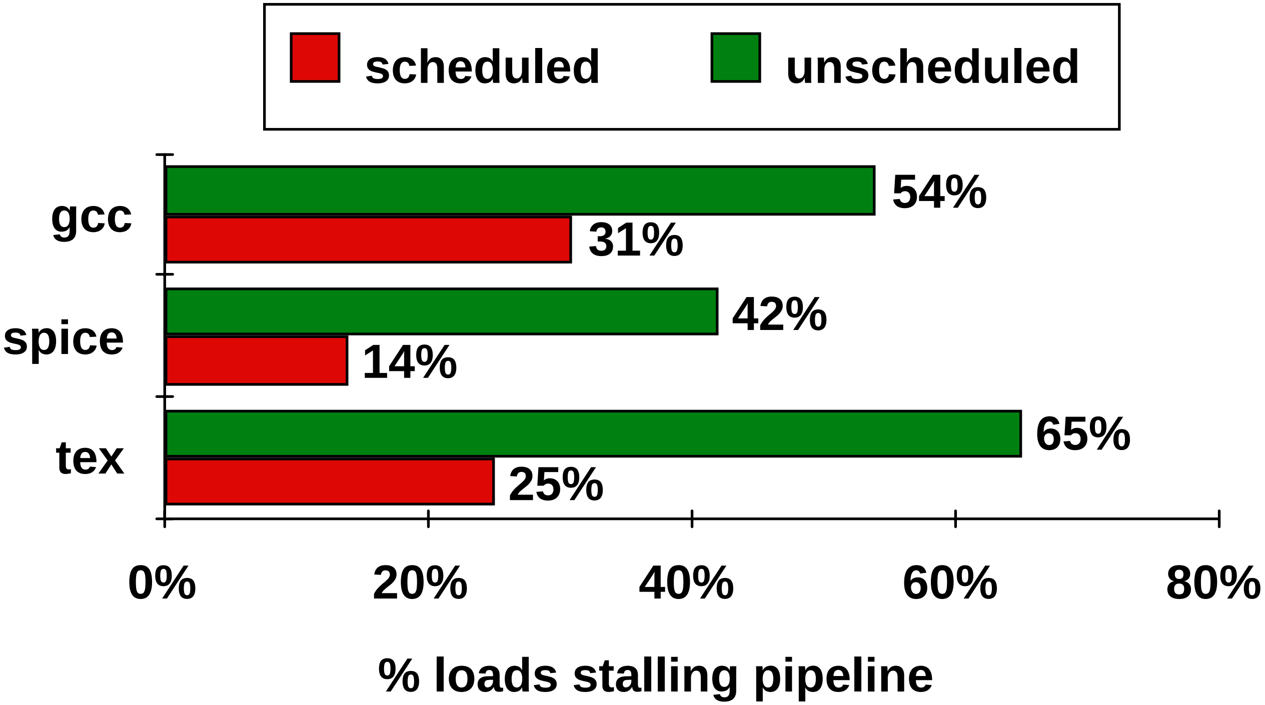
- What about data dependence on loads?

  R1 <- R4 + R5
  R2 <- Mem[R2 + I]
  R3 <- R2 + R1

=> "Delayed Loads"



Compiler Avoiding Load Stalls:

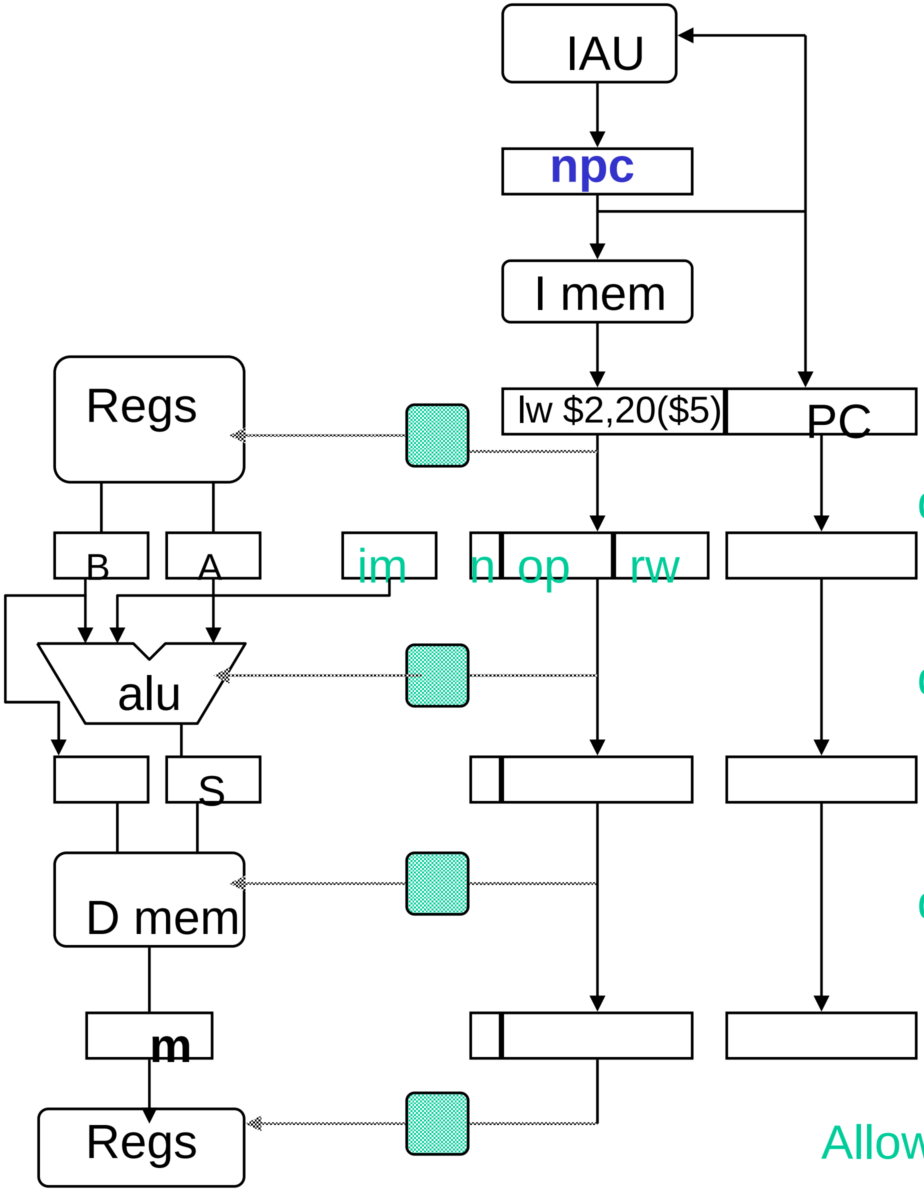


What about Interrupts, Traps, Faults?

- **External Interrupts:**
  - Allow pipeline to drain,
  - Load PC with interupt address
- **Faults (within instruction, restartable)**
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state

Refer to MIPS solution

Exception Handling



detect bad instruction address

detect bad instruction

detect overflow

detect bad data address

Allow exception to take effect

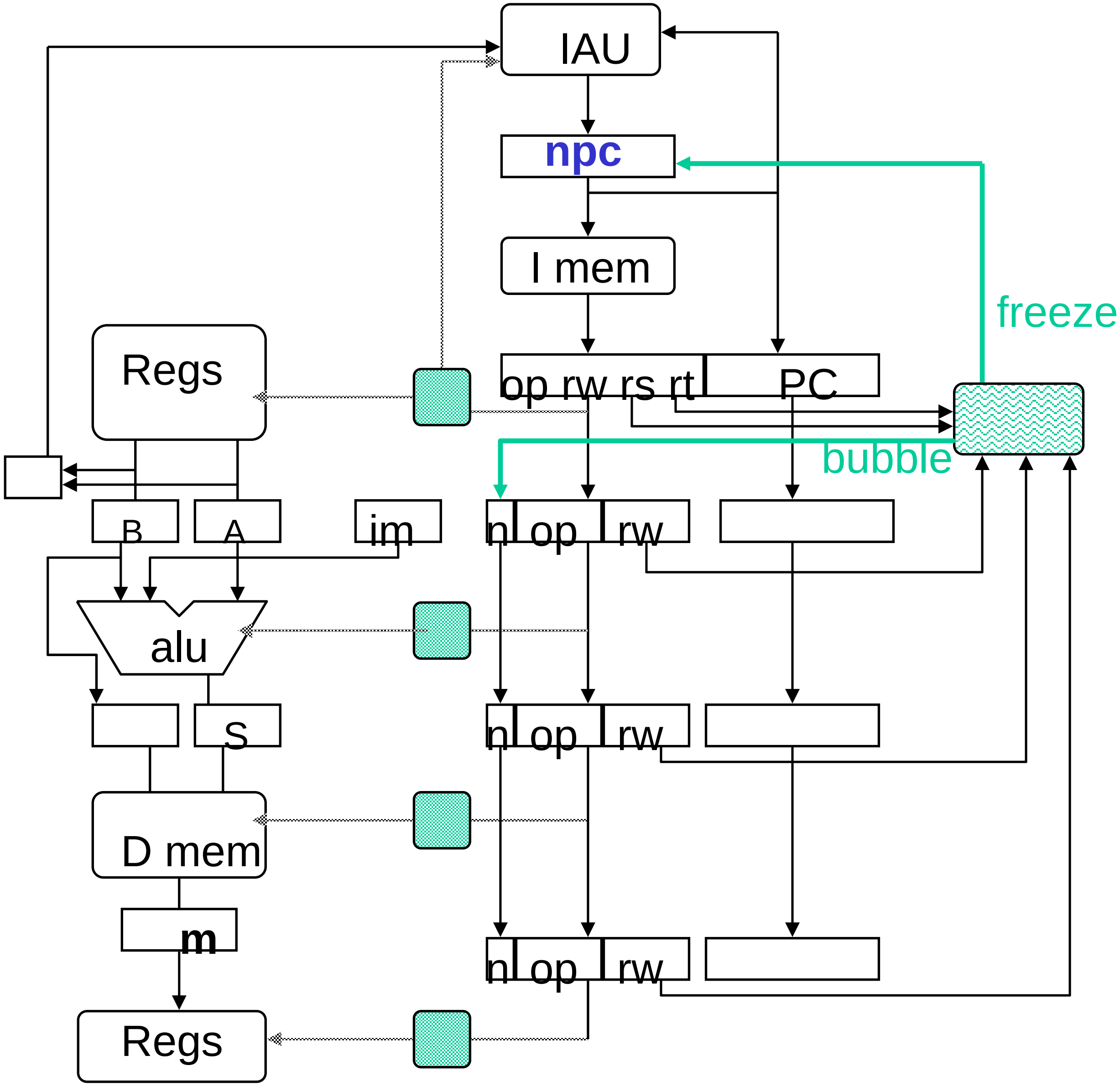
Exception Problem

- Exceptions/Interrupts: 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

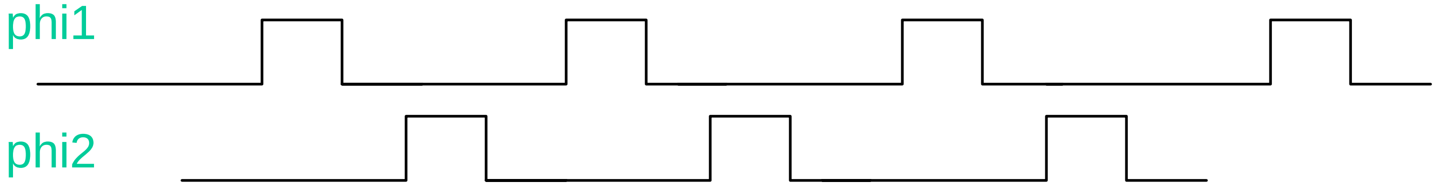
Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction , check last stage
- Solution 2: interrupt ASAP, restart everything incomplete

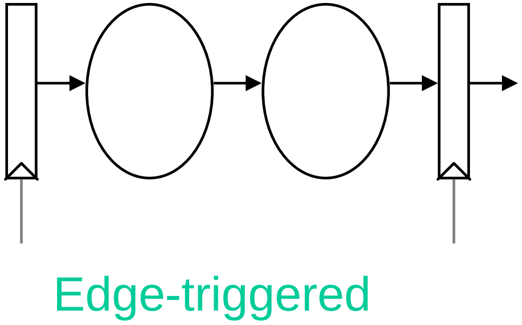
Resolution: Freeze above & Bubble Below

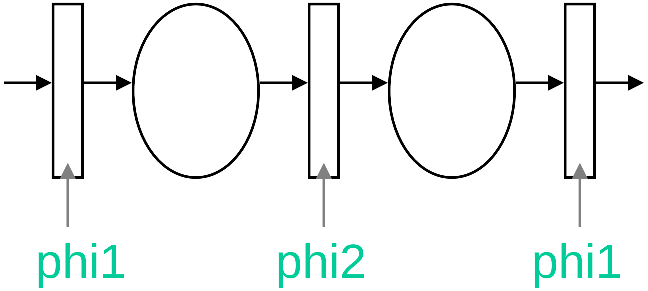


FYI: MIPS R3000 clocking discipline



- **2-phase non-overlapping clocks**
- **Pipeline stage is two (level sensitive) latches**

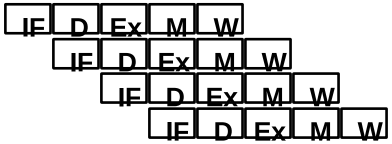




Issues in Pipelined Design

Method | **Limitation**

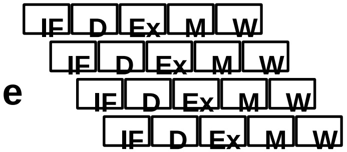
° Pipelining



Issue rate, FU stalls, FU depth

° Super-pipeline

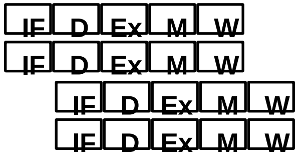
- Issue one instruction per (fast) cycle
- ALU takes multiple cycles



Clock skew, FU stalls, FU depth

° Super-scalar

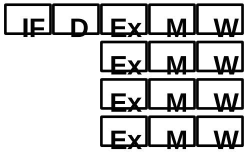
- Issue multiple scalar instructions per cycle



Hazard resolution

° VLIW (“EPIC”)

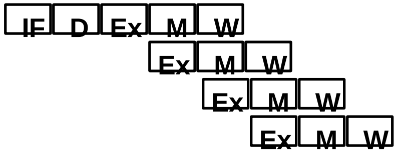
- Each instruction specifies multiple scalar operations
- Compiler determines parallelism



Packing

° Vector operations

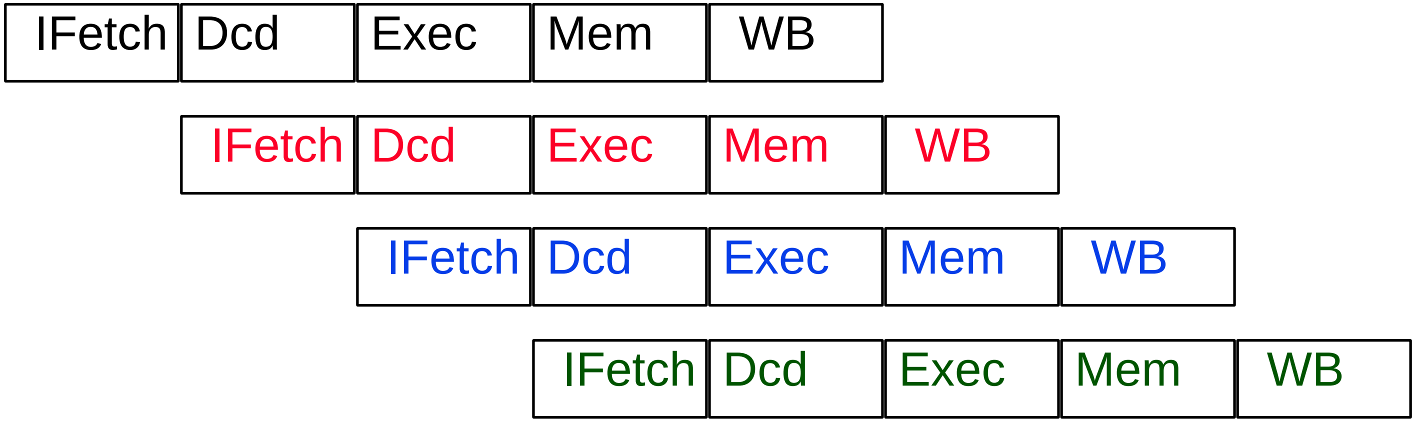
- Each instruction specifies series of identical operations



Applicability

Is CPI = 1 for our pipeline?

- **Remember that CPI is an “Average # cycles/inst**



IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

IFetch Dcd Exec Mem WB

- **CPI here is 1, since the average throughput is 1 instruction every cycle.**
- **What if there are stalls or multi-cycle execution?**
- **Usually CPI > 1. How close can we get to 1??**

Computation of CPI when Pipeline Stalls are Present

$$CPI = CPI_{base} CPI_{stall}$$

$$CPI_{stall} = STALL_{type1} \times freq_{type1} + STALL_{type2} \times freq_{type2}$$

- **Start with Base CPI**
- **Add stalls**

- **Suppose:**
  - **CPI_{base} = 1 ;**
  - **$freq_{branch}$ = 20%, $freq_{load}$ = 30% ;**
  - **Suppose branches always cause 1 cycle stall;**
  - **Loads cause a 100 cycle stall 1% of time;**
  - **Then:** CPI = 1 + (1 x 0.20) + (100 x 0.30 x 0.01) = 1.5
- **Multicycle? Could treat as:**
  - CPI_{stall} = (CYCLES - CPI_{base}) x $freq_{inst}$

FP Loop: Where Are the Hazards?

```
Loop:  LD    F0, 0(R1)      ;F0 = vector element
       ADDD  F4, F0, F2     ;add scalar from F2
       SD    0(R1), F4      ;store result
       SUBI  R1, R1, 8      ;decrement pointer 8B (DW)
       BNEZ  R1, Loop       ;branch R1 != zero
       NOP                  ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- **Where are the stalls?**

FP Loop Showing Stalls

```
1 Loop: LD     F0, 0(R1)    ;F0=vector element
2       stall
3       ADDD   F4, F0, F2   ;add scalar in F2
4       stall
5       stall
6       SD     0(R1), F4    ;store result
7       SUBI   R1, R1, 8    ;decrement pointer 8B (DW)
8       BNEZ   R1, Loop     ;branch R1!=zero
9       stall               ;delayed branch slot
```

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- **9 clocks: Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop: LD    F0,0(R1)
2       stall
3       ADDD  F4,F0,F2
4       SUBI  R1,R1,8
5       BNEZ  R1,Loop   ;delayed branch
6       SD    8(R1),F4  ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

Instruction producing result	*Instruction using result*	*Latency in clock cycles*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks: Unroll loop 4 times code to make faster?

Unroll Loop Four Times (straightforward way)

Rewrite loop to minimize stalls?

1 cycle stall

2 cycles stall

```
1  Loop:LD     F0,0(R1)
2        ADDD   F4,F0,F2
3        SD     0(R1),F4      ;drop SUBI & BNEZ
4        LD     F6,-8(R1)
5        ADDD   F8,F6,F2
6        SD     -8(R1),F8     ;drop SUBI & BNEZ
7        LD     F10,-16(R1)
8        ADDD   F12,F10,F2
9        SD     -16(R1),F12   ;drop SUBI & BNEZ
10       LD     F14,-24(R1)
11       ADDD   F16,F14,F2
12       SD     -24(R1),F16
13       SUBI   R1,R1,#32     ;alter to 4*8
14       BNEZ   R1,LOOP
15       NOP
```

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop:LD     F0, 0(R1)
2      LD     F6, -8(R1)
3      LD     F10, -16(R1)
4      LD     F14, -24(R1)
5      ADDD   F4, F0, F2
6      ADDD   F8, F6, F2
7      ADDD   F12, F10, F2
8      ADDD   F16, F14, F2
9      SD     0(R1), F4
10     SD     -8(R1), F8
11     SD     -16(R1), F12
12     SUBI   R1, R1, #32
13     BNEZ   R1, LOOP
14     SD     8(R1), F16   ; 8-32 = -24
```

- **What assumptions made when moved code?**
  - OK to move store past SUBI even though changes register
  - OK to move loads before stores: get right data?
  - When is it safe for compiler to do such changes?

14 clock cycles, or 3.5 per iteration

When safe to move instructions?

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Two main variations: Superscalar and VLIW**
- **Superscalar: varying no. instructions/cycle (1 to 6)**
  - Parallelism and dependencies determined/resolved by HW;
  - IBM PowerPC 604, Sun UltraSparc, DEC Alpha 21164, HP 7100.
- **Very Long Instruction Words (VLIW): fixed number of instructions (16); parallelism determined by compiler:**
  - pipeline is exposed;
  - compiler must schedule delays to get right result.
- **Explicit Parallel Instruction Computer (EPIC) [Intel]**
  - 128 bit packets containing 3 instructions (can execute sequentially);
  - Can link 128 bit packets together to allow more parallelism;
  - Compiler determines parallelism;
  - HW checks dependencies and forwards/stalls.

Getting CPI < 1: Issuing Multiple Instructions/Cycle – II

- **Superscalar DLX: 2 instructions, 1 FP & 1 anything else**

– Fetch 64-bits/clock cycle; Int on left, FP on right

– Can only issue 2nd instruction if 1st instruction issues

– More ports for FP registers to do FP load & FP op in a pair

Type	***Pipe Stages***						
Int. instruction	**IF**	**ID**	**EX**	**MEM**	**WB**		
FP..instruction	**IF**	**ID**	**EX**	**MEM**	**WB**		
Int..instruction		**IF**	**ID**	**EX**	**MEM**	**WB**	
FP..instruction		**IF**	**ID**	**EX**	**MEM**	**WB**	
Int. instruction			**IF**	**ID**	**EX**	**MEM**	**WB**
FP instruction			**IF**	**ID**	**EX**	**MEM**	**WB**

- **1 cycle load delay expands to 3 instructions in SS**
  - instruction in right half can’t use it, nor instructions in next slot

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	**LD F0, 0(R1)**		**1**
	LD F6, -8(R1)		**2**
	LD F10, -16(R1)	**ADDD F4, F0, F2**	**3**
	LD F14, -24(R1)	**ADDD F8, F6, F2**	**4**
	LD F18, -32(R1)	**ADDD F12, F10, F2**	**5**
	SD 0(R1), F4	**ADDD F16, F14, F2**	**6**
	SD -8(R1), F8	**ADDD F20, F18, F2**	**7**
	SD -16(R1), F12		**8**
	SD -24(R1), F16		**9**
	SUBI R1, R1, #40		**10**
	BNEZ R1, LOOP		**11**
	SD -32(R1), F20		**12**

- **Unrolled 5 times to avoid delays (+1 due to SS)**
- **12 clocks, or 2.4 clocks per iteration**

Limits of Superscalar

- **While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:**
  - Exactly 50% FP operations;
  - No hazards.
- **If more instructions issue at same time, *greater difficulty of decode and issue.***
  - Even 2-scalar ⇒ examine 2 opcodes, 6 register specifiers, and decide if 1 or 2 instructions can issue.
- **VLIW: tradeoff instruction space for simple decoding:**
  - The long instruction word has room for many operations;
  - By definition, all the operations the compiler puts in the long instruction word can execute in parallel;
  - *e.g.*, 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch.
    - 16 to 24 bits per field ⇒ 7×16 or 112 bits to 7×24 or 168 bits wide.
  - Need compiling technique that schedules across *several branches.*

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP op. 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4, F0, F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12, F10, F2	ADDD F16,F14,F2		4
		ADDD F20, F18, F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28, F26, F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#48	8
SD -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration

Need more registers in VLIW(EPIC => 128int + 128FP)

Summary

- **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? HAZARDS!**
  - 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
- **Pipelines pass control information down the pipe just as data moves down pipe**
- **Forwarding/Stalls handled by local control**
- **Exceptions stop the pipeline**

Summary

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- Exceptions stop the pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism

Summary

- **Hazards limit performance**
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- **Data hazards must be handled carefully:**
  - RAW data hazards handled by forwarding
  - WAW and WAR hazards don’t exist in 5-stage pipeline
- **MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)**
- **Exceptions in 5-stage pipeline recorded when they occur, but acted on only at WB (end of MEM) stage**
  - Must flush all previous instructions
- **More performance from deeper pipelines, parallelism**