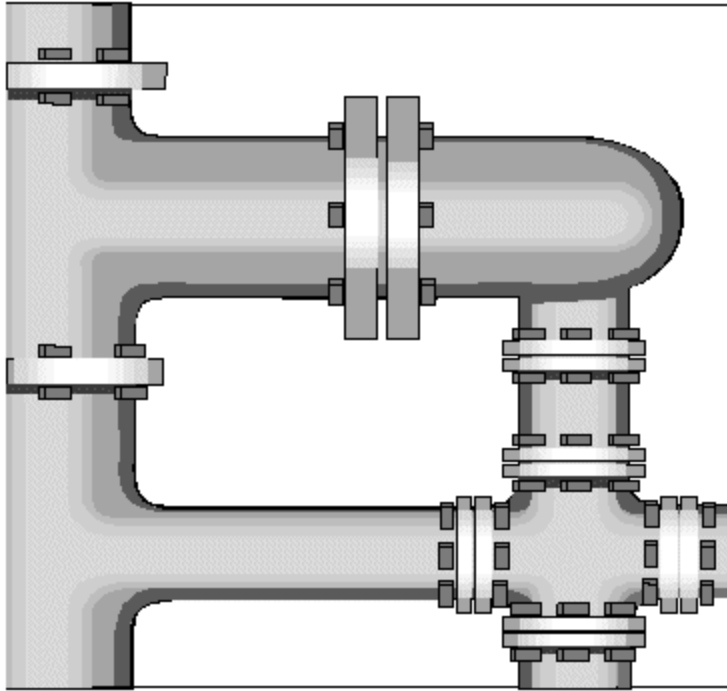# Pipelined CPUs

Where are the registers?

Second Quiz on Friday. Covers lectures 8-14. Open book, open note, no computers or calculators.

Study Chapter 4 of Text

Review of CPU Performance

$$MIPS = \frac{Freq}{CPI}$$

MIPS = Millions of Instructions/Second

Freq = Clock Frequency, MHz

CPI = Clocks per Instruction

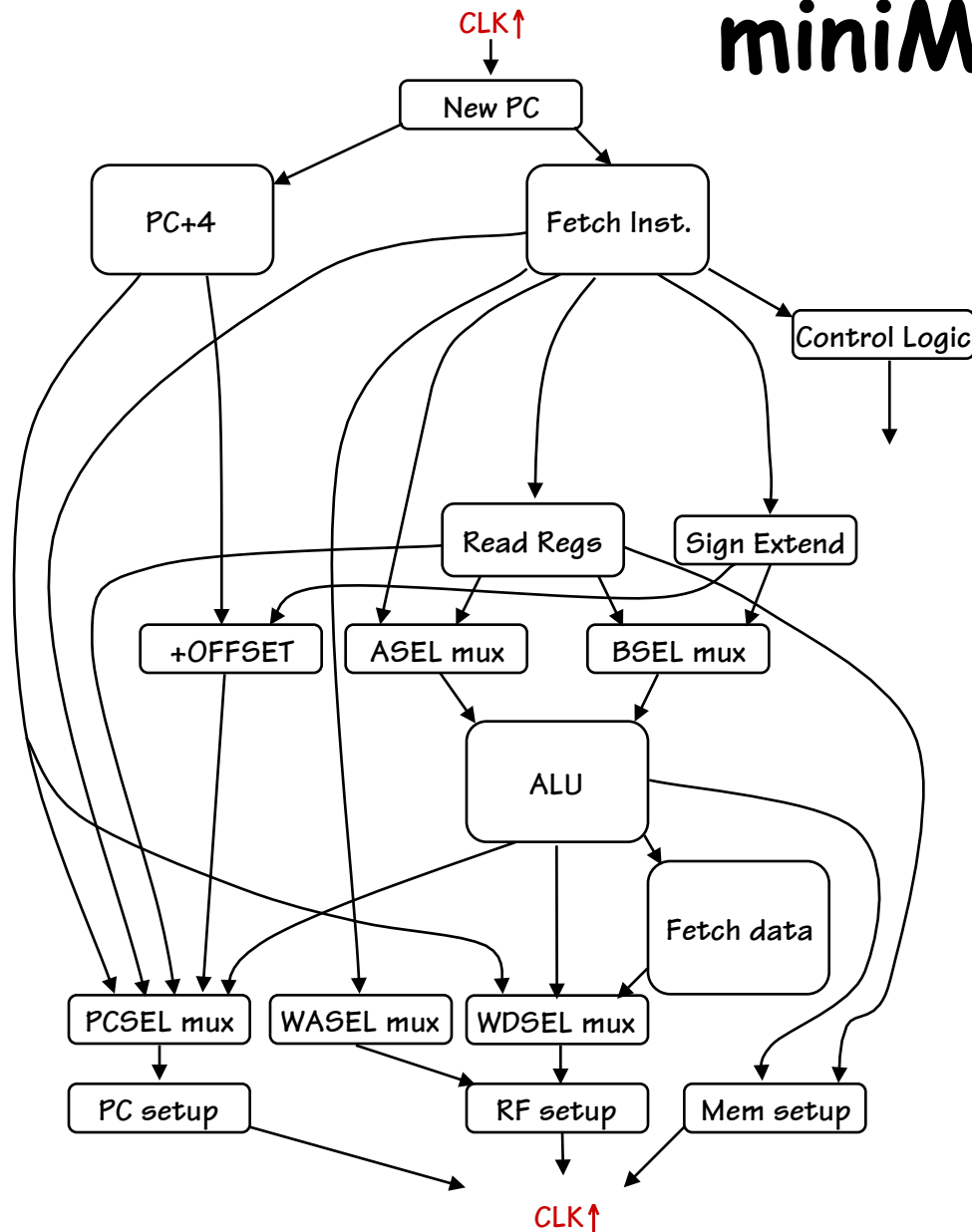To Increase MIPS:

    1. DECREASE CPI.

      - RISC *simplicity* reduces CPI to 1.0.

      - CPI *below* 1.0? State-of-the-art multiple instruction issue

    2. INCREASE Freq.

      - Freq limited by delay along longest combinational path; hence

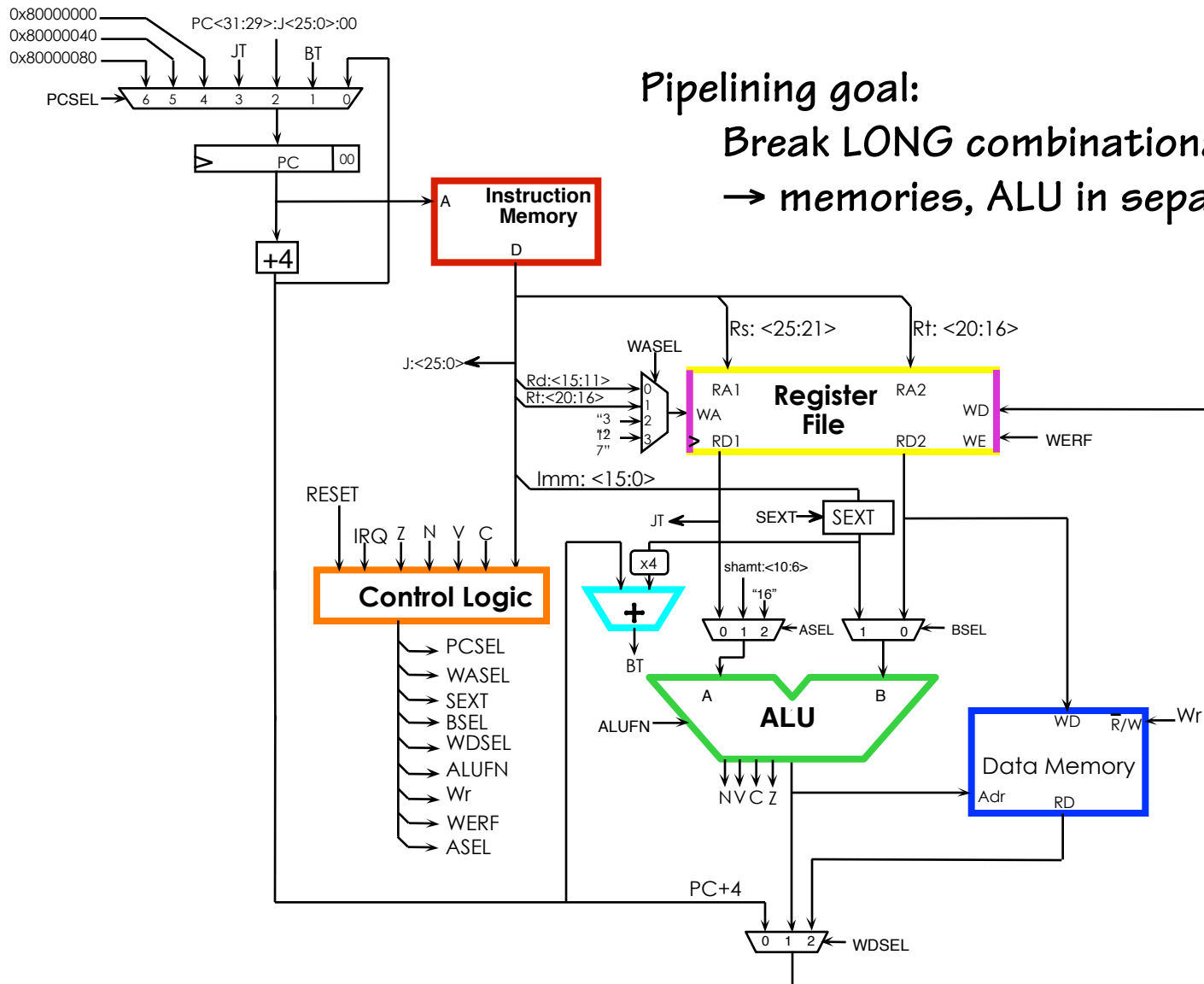   - *PIPELINING* is the key to improving performance.

miniMIPS Timing

The diagram on the left illustrates the Data Flow of miniMIPS

Wanted: longest path

Complications:

- some apparent paths aren't "possible"

- functional units have variable execution times (eg, ALU)

- time axis is not to scale (eg, $t_{PD,MEM}$ is very big!)
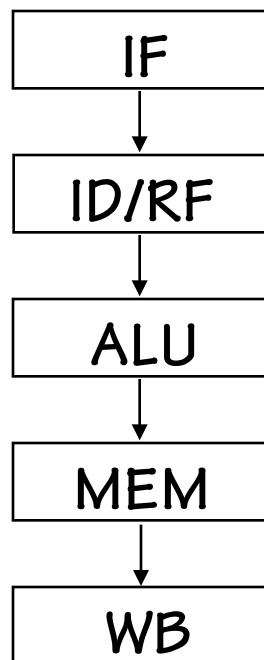
Where Are the Bottlenecks?

Pipelining goal:

Break LONG combinational paths

→ memories, ALU in separate stages

Ultimate Goal: 5-Stage Pipeline

GOAL: Maintain (nearly) 1.0 CPI, but increase clock speed to barely include slowest components (mems, regfile, ALU)

APPROACH: structure processor as 5-stage pipeline:

IF
ID/RF
ALU
MEM
WB

Instruction Fetch stage: Maintains PC, fetches one instruction per cycle and passes it to

Instruction Decode/Register File stage: Decode control lines and select source operands
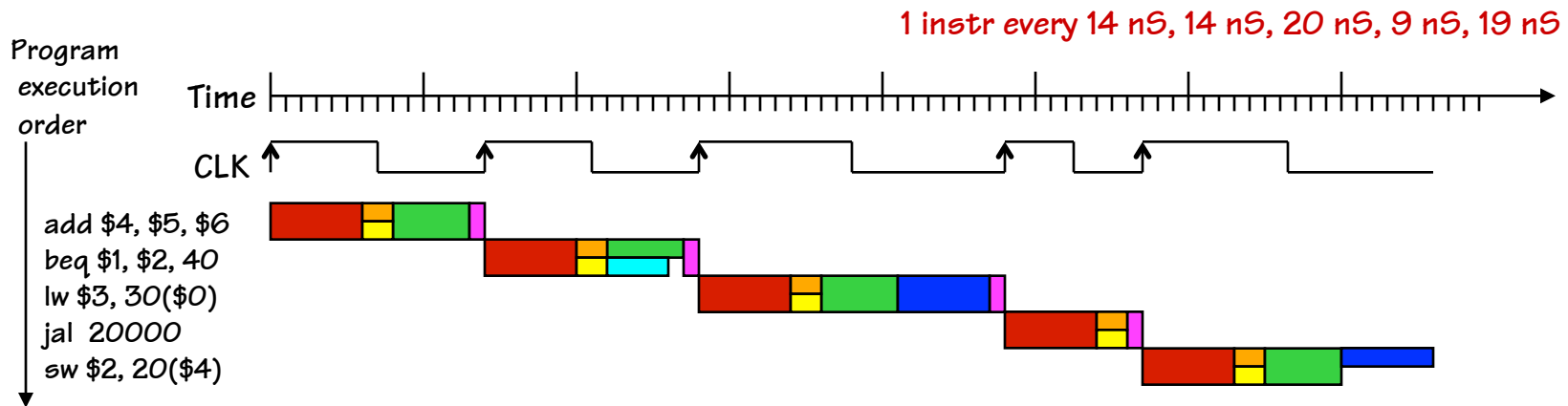
ALU stage: Performs specified operation, passes result to

Memory stage: If it's a lw, use ALU result as an address, pass mem data (or ALU result if not lw) to

Write-Back stage: writes result back into register file.

miniMIPS Timing

Different instructions use various parts of the data path.

1 instr every 14 nS, 14 nS, 20 nS, 9 nS, 19 nS

Program execution order

Time

CLK

add $4, $5, $6
beq $1, $2, 40
lw $3, 30($0)
jal 20000
sw $2, 20($4)

6 nS	Instruction Fetch
2 nS	Instruction Decode
2 nS	Register Prop Delay
5 nS	ALU Operation
4 nS	Branch Target
6 nS	Data Access
1 nS	Register Setup

This is an example of a "Asynchronous Globally-Timed" control strategy (see Lecture 16). Such a system would vary the clock period based on the instruction being executed. This leads to complicated timing generation, and, in the end, slower systems, since it is not very compatible with pipelining!

Uniform miniMIPS Timing

With a fixed clock period, we have to allow for the worse case.
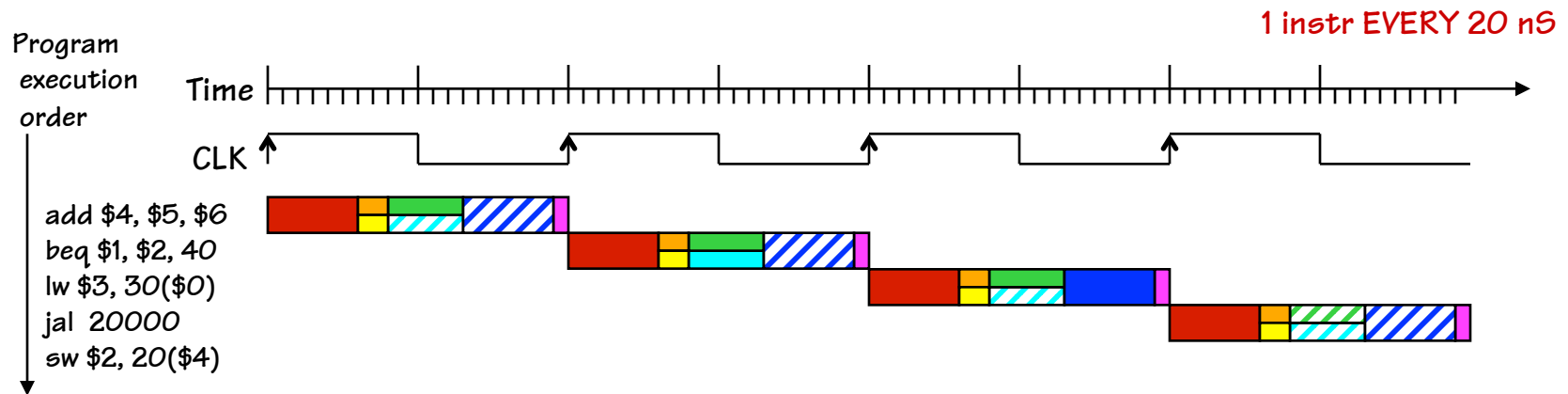
1 instr EVERY 20 nS

Program execution order

Time

CLK

add $4, $5, $6
beq $1, $2, 40
lw $3, 30($0)
jal 20000
sw $2, 20($4)

6 nS	■ Instruction Fetch
2 nS	■ Instruction Decode
2 nS	■ Register Prop Delay
5 nS	■ ALU Operation
4 nS	■ Branch Target
6 nS	■ Data Access
1 nS	■ Register Setup

By accounting for the "worse case" path (i.e. allowing time for each possible combination of operations) we can implement a "Synchronous Globally -Timed" control strategy. This simplifies timing generation, enforces a uniform processing order, and allows for pipelining!

Isn't the net effect just a slower CPU?

Step 1: A 2-Stage Pipeline

0x80000000
0x80000040
0x80000080

PC<31:29>:J<25:0>:00

JT   BT

PCSEL → | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

> PC 00

+4

> PC^EXE 00   > IR^EXE

IF

EXE

J:<25:0>

WASEL   Rs: <25:21>   Rt: <20:16>

Rd:<15:11> → 0
Rt:<20:16> → 1
"3" → 2
"2" → 3
"7"

RA1 **Register File** RA2

WA   WD

RD1   RD2   WE ← WERF

Imm: <15:0>

RESET

IRQ Z N V C

JT   SEXT → SEXT

x4   shamt:<10:6>

Control Logic

+   "16"

| 0 | 1 | 2 | ← ASEL | 1 | 0 | ← BSEL

BT

→ PCSEL
→ WASEL
→ SEXT
→ BSEL
→ WDSEL
→ ALUFN
→ Wr
→ WERF
→ ASEL

A   **ALU**   B

ALUFN →

N V C Z

WD   R̄/W ← Wr

Data Memory

Adr   RD

PC+4

| 0 | 1 | 2 | ← WDSEL

IR stands for "Instruction Register". The superscript "EXE" denotes the pipeline stage, in which the PC and IR are used.

2-Stage Pipe Timing

Improves performance by increasing instruction throughput.
Ideal speedup is number of pipeline stages in the pipeline.

Program
execution
order

Time

CLK

add $4, $5, $6
beq $1, $2, 40
lw $3, 30($0)
jal 20000
sw $2, 20($4)

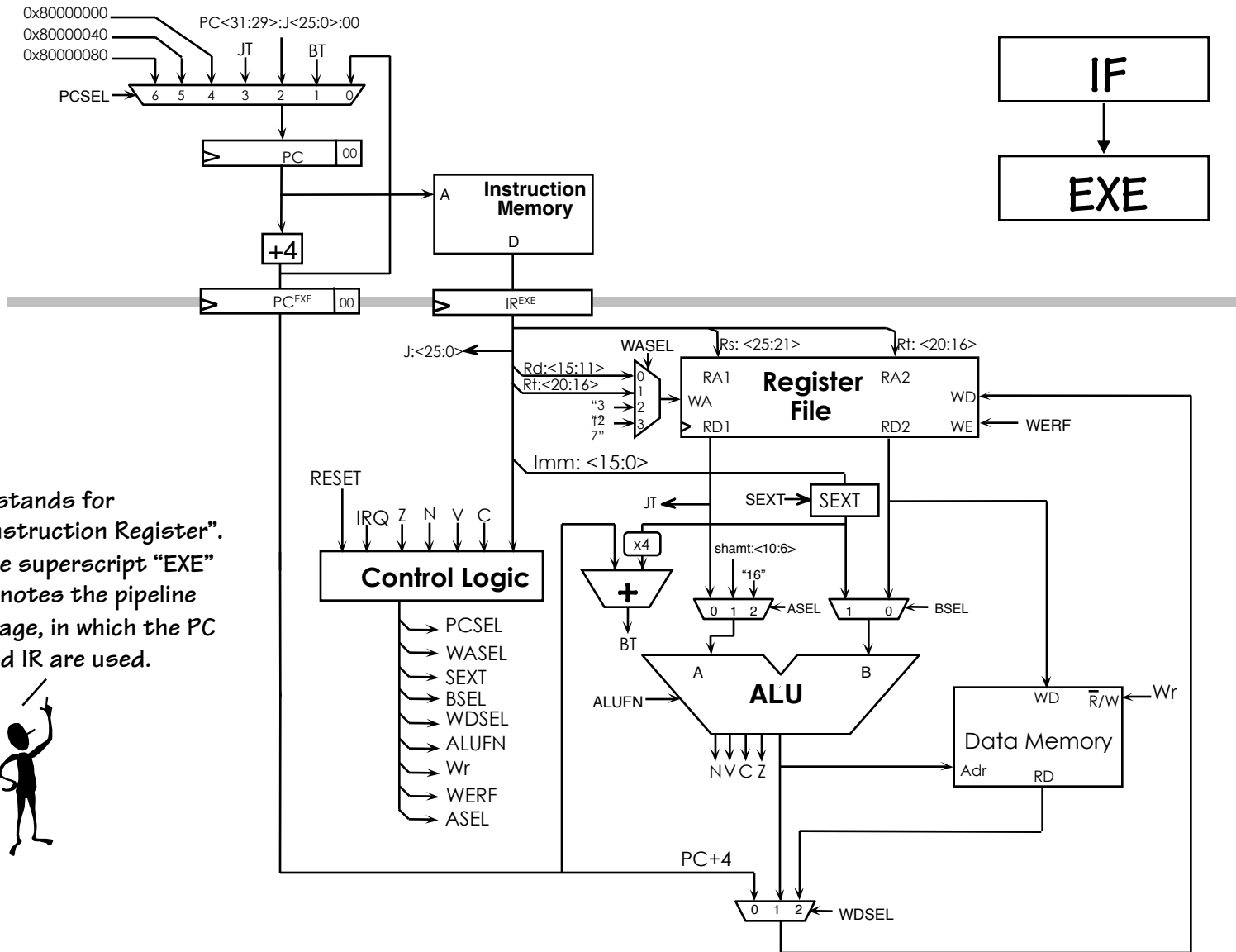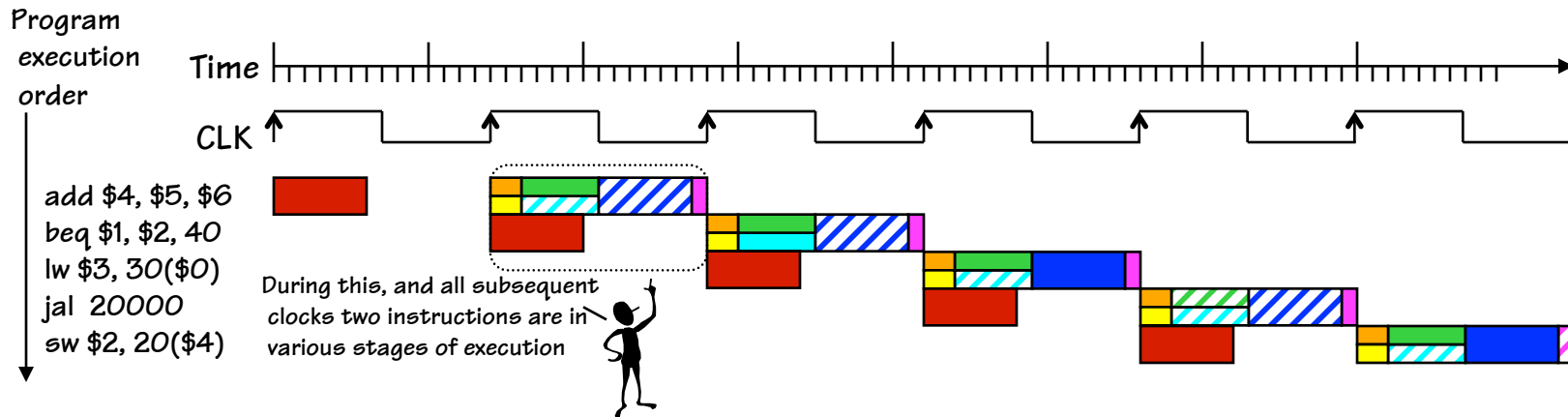During this, and all subsequent clocks two instructions are in various stages of execution

6 nS — Instruction Fetch
2 nS — Instruction Decode
2 nS — Register Prop Delay
5 nS — ALU Operation
4 nS — Branch Target
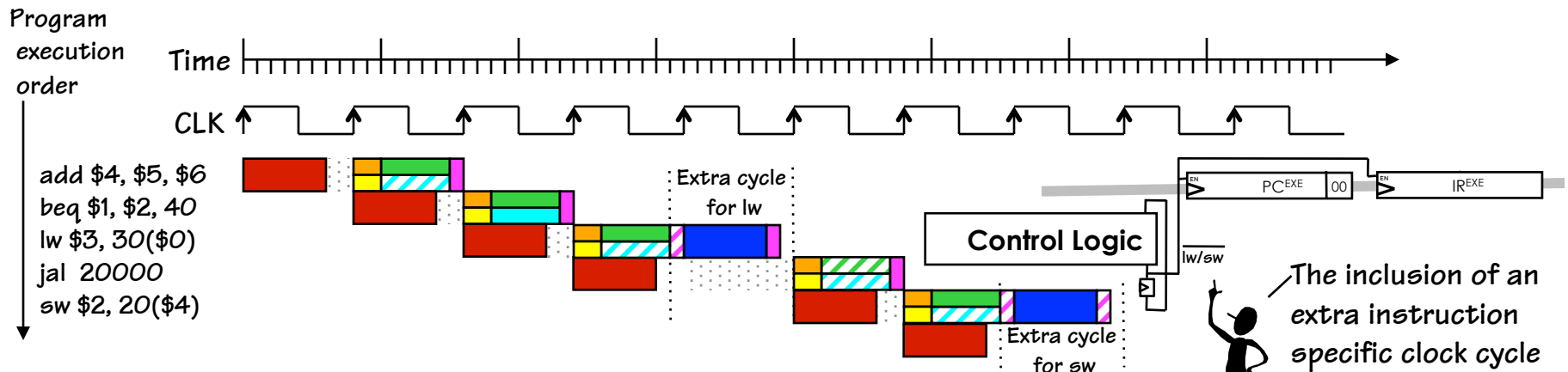6 nS — Data Access
1 nS — Register Setup

By partitioning each instruction cycle into a "fetch" stage and an "execute" stage, we get a simple pipeline. Why not include the Instruction-Decode/Register-Access time with the Instruction Fetch? You could. But this partitioning allows for a useful variant with 2-cycle loads and stores.

Latency?
2 Clock periods = 2*14 nS
Throughput?
1 instr per 14 nS

2-Stage w/2-Cycle Loads & Stores

Further improves performance, with slight increase in control complexity.
Some 1st generation (pre-cache) RISC processors used this approach.

Program execution order

Time

CLK

add $4, $5, $6
beq $1, $2, 40
lw $3, 30($0)
jal 20000
sw $2, 20($4)

Extra cycle for lw

Extra cycle for sw

Control Logic

lw/sw

PC^{EXE} 00   IR^{EXE}
EN   EN

The inclusion of an extra instruction specific clock cycle within a normal pipeline is called "inserting a bubble".

6 nS	Instruction Fetch
2 nS	Instruction Decode
2 nS	Register Prop Delay
5 nS	ALU Operation
4 nS	Branch Target
6 nS	Data Access
1 nS	Register Setup

The clock rate of this variant is nearly twice that of our original design. Does that mean it is twice as fast?
Not likely. In practice, as many as 30% of instructions access memory. Thus, the effective speed up is:

$$speed\ up = \frac{old\ clock\ period}{new\ clock\ period\,(0.7+2*0.3)}$$

$$= \frac{20}{8(1.3)} = 1.923$$

Clock: 8 nS!

2-Stage Pipelined Operation

Consider a sequence
of instructions:

```
...
addi        $t2,$t1,1
xor         $t2,$t1,$t2
sltiu       $t3,$t2,1
srl         $t2,$t2,1
...
```

Recall "Pipeline Diagrams" from Lecture 16.

Executed on our 2-stage pipeline:

It can't be this easy!?

TIME (cycles) →

Pipeline ↓

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	xor	sltiu	srl	...		
EXE		addi	xor	sltiu	srl	...	
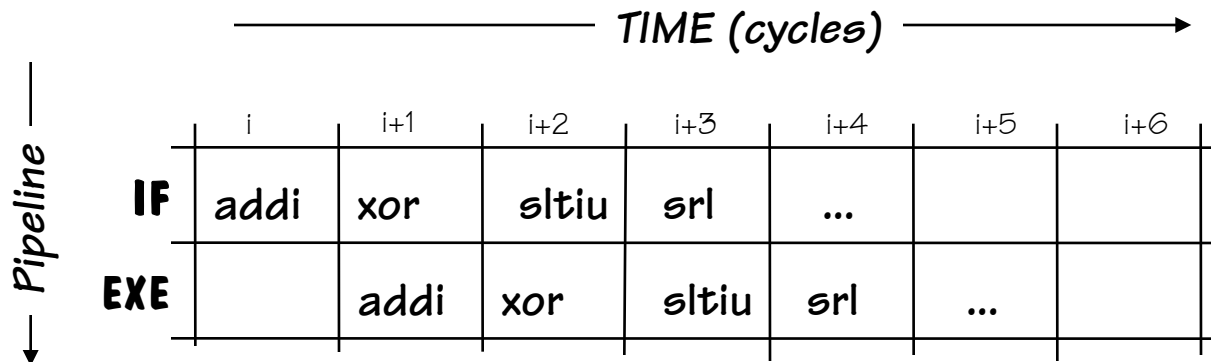
Pipeline Control Hazards

BUT consider instead:

```
loop:  add   $t1,$t1,$t0
       srl   $t2,$t2,1
       bne   $t2,$0,loop
       andi  $t0,$t2,1
       ...
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	add	srl	bne	andi	...		
EXE		add	srl	bne	?	...	

This is the cycle where the branch decision is made... but we've already fetched the following instruction which should be executed only if branch is not taken!

Pipelining HAZARDS are situations where the next instruction cannot execute in the next clock cycle. There are two forms of hazards, CONTROL and STRUCTURAL.

Branch Delay Slots

PROBLEM: One (or more) instructions following a branch are fetched before the branch decision is made (to take, or not to take).

POSSIBLE SOLUTIONS:

1. Make hardware "annul" the instructions following taken branches, e.g., by disabling WERF and WR.

Solution 2 implies breaking the sequential semantics of the ISA. Logically, the branch takes place after instructions in the DELAY SLOTS are executed.

2. "Program around it". Either

  a) Follow each BNE/BEQ with a NOP instruction; or

  b) Make compiler clever enough to move USEFUL instructions into the branch delay slots

    i. Always execute instructions in delay slots

    ii. Conditionally execute instructions in delay slots
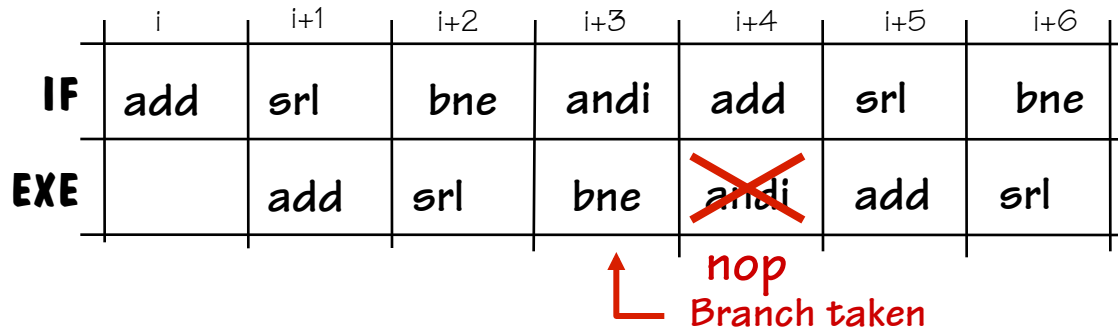
- Delay slots also apply to jump instructions: `j`, `jal`, and `jr`

Branch Solution 1

Make the hardware annul
instructions in the
branch delay slots of a
taken branch.

```
loop:  add   $t1,$t1,$t0
       srl   $t2,$t2,1
       bne   $t2,$0,loop
       andi  $t0,$t2,1
       ...
```

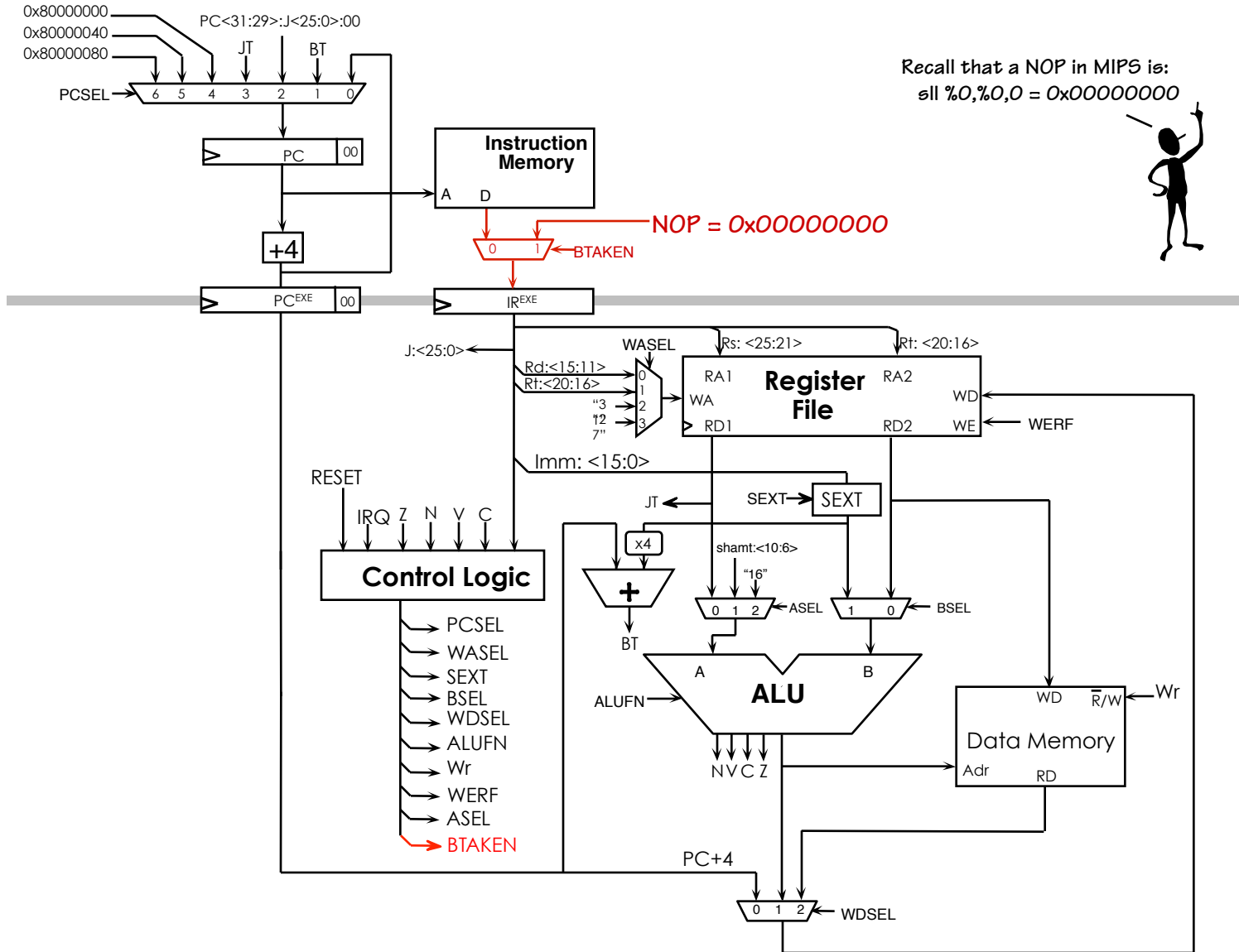|	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	add	srl	bne	andi	add	srl	bne
EXE		add	srl	bne	~~andi~~	add	srl

nop
Branch taken

Pros: Programs run identically on both unpipelined and pipelined hardware

Cons: in SPEC benchmarks 14% of instructions are *taken branches* →

14/114 = 12% of total cycles are annulled

Branch Annulment Hardware

Recall that a NOP in MIPS is:
sll %0,%0,0 = 0x00000000

NOP = 0x00000000

Branch Alternative 2a

Always fill branch delay
slots with NOP
instructions.
Worse than H/W annulment.
NOPs get executed whether
branches are taken or not.

```
loop:  add   $t1,$t1,$t0
       srl   $t2,$t2,1
       bne   $t2,$0,loop
       nop
       andi  $t0,$t2,1
       ...
```

Maybe I could
find something
useful to do in
that instruction
slot

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	add	srl	bne	nop	add	srl	bne
EXE		add	cmp	bne	nop	add	cmp

↑ Branch taken
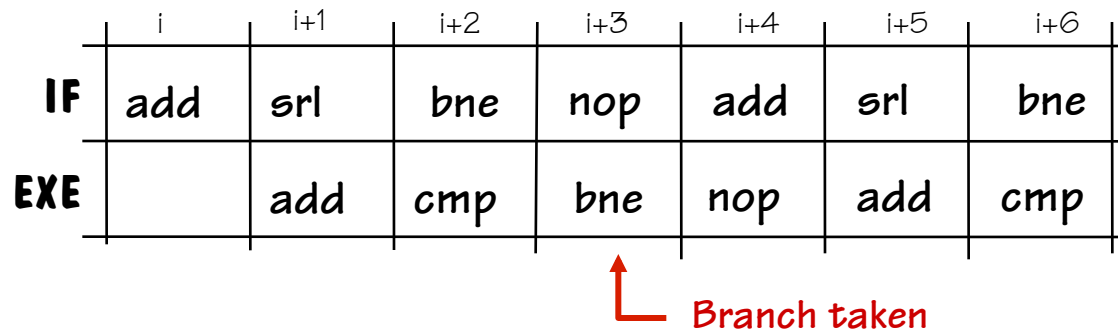
Pros: Does not require H/W modifications, only compiler changes

Cons: NOPs make code longer; >12% of cycles spent executing NOPs

Branch Alternative 2b(i)

Put USEFUL instructions in the branch delay slots; remember they will be executed whether the branch is taken or not

```
            srl   $t2,$t2,1
loop:       add   $t1,$t1,$t0
            bne   $t2,$0,loop
            srl   $t2,$t2,1
            ...
```

Effectively a NOP if the branch is not taken.
(if ($t2 == 0) then $t2 >> 1 == 0)

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	srl	add	bne	srl	add	bne	srl
EXE		srl	add	bne	srl	add	bne

⌐ Branch taken

However, finding an instruction that behaves like a NOP when not taken can be tricky,

Pros: only one "extra" instruction is executed (on last iteration)

Cons: finding "useful" instructions that should *always* be executed is difficult; clever rewrite may be required. Program executes differently on unpipelined implementation.

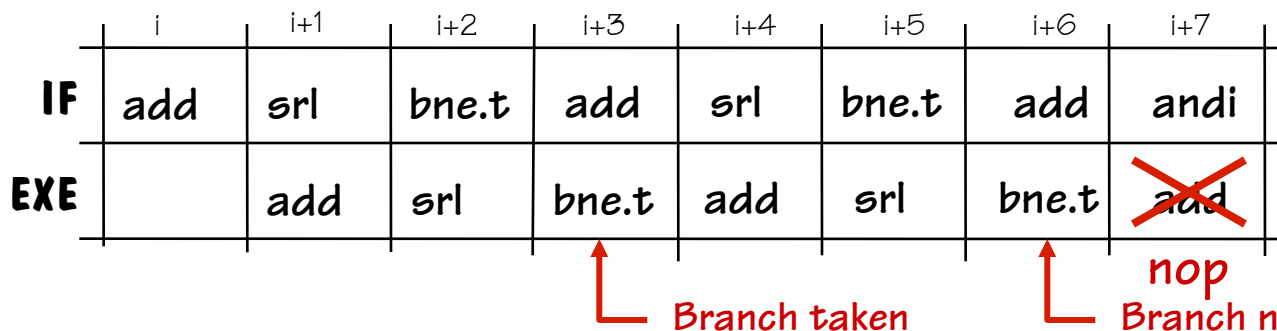This is the standard approach for pipelined MIPS implementations

Branch Alternative 2b(ii)

Put USEFUL instructions in the branch delay slots; annul them if branch *doesn't behave as predicted*

```
          add      $t1,$t1,$t0
loop:     srl      $t2,$t2,1
          bne.t    $t2,$0,loop
          add      $t1,$t1,$t0
          andi     $t0,$t2,1
          ...
```

The ".t" suffix, implies a new instruction variant "Branch if not equal, while executing the delay slot if taken." Likewise, we could add a ".n" variant for the "execute if not taken" case. H/W annuls the "opposite" case.

	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7
IF	add	srl	bne.t	add	srl	bne.t	add	andi
EXE		add	srl	bne.t	add	srl	bne.t	~~add~~

nop

Branch taken

Branch not taken

Pros: only one instruction is annulled (on last iteration); about 70% of branch delay slots can be filled with useful instructions

Cons: Program executes differently on naïve unpipelined implementation; difficult to utilize with more than one delay slot.

Architectural Issue:
Branch Decision Timing

The number of branch delay slots is determined by "where" in the pipeline the "branch decision" is made relative to where the next instruction is fetched.
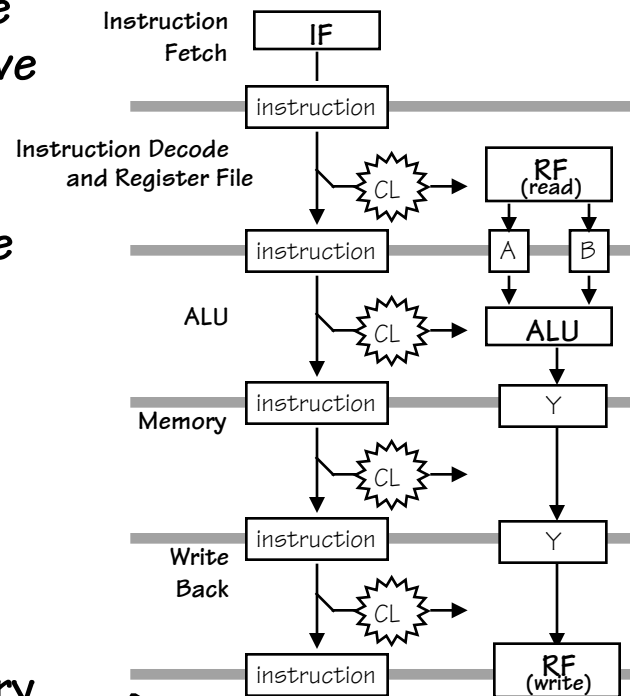
Consider the 5-stage miniMIPS pipeline shown on the right.

Where is the branch decision resolved?

```
beq rs,rt,offset
```
if (Reg[rs] == Reg[rt])
  PC ← PC + 4 + 4*SEXT(offset)

The decision is based on the ALU's Z-flag, which is determined at the very end of the ALU stage, nearly 2 clocks after the instruction fetch. Therefore, a naïve 5-stage pipelined miniMIPS implementation has at least TWO branch delay slots.

Instruction Fetch — IF

instruction

Instruction Decode and Register File — CL → RF (read)

instruction — A   B

ALU — CL → ALU

instruction — Y

Memory — CL

instruction — Y

Write Back — CL

instruction — RF (write)

Is there any way miniMIPS' could make its branch decision sooner? We only need to support BNE and BEQ, since we choose to trap and emulate the more complicated branch instructions.

Early Branch Decision Hardware
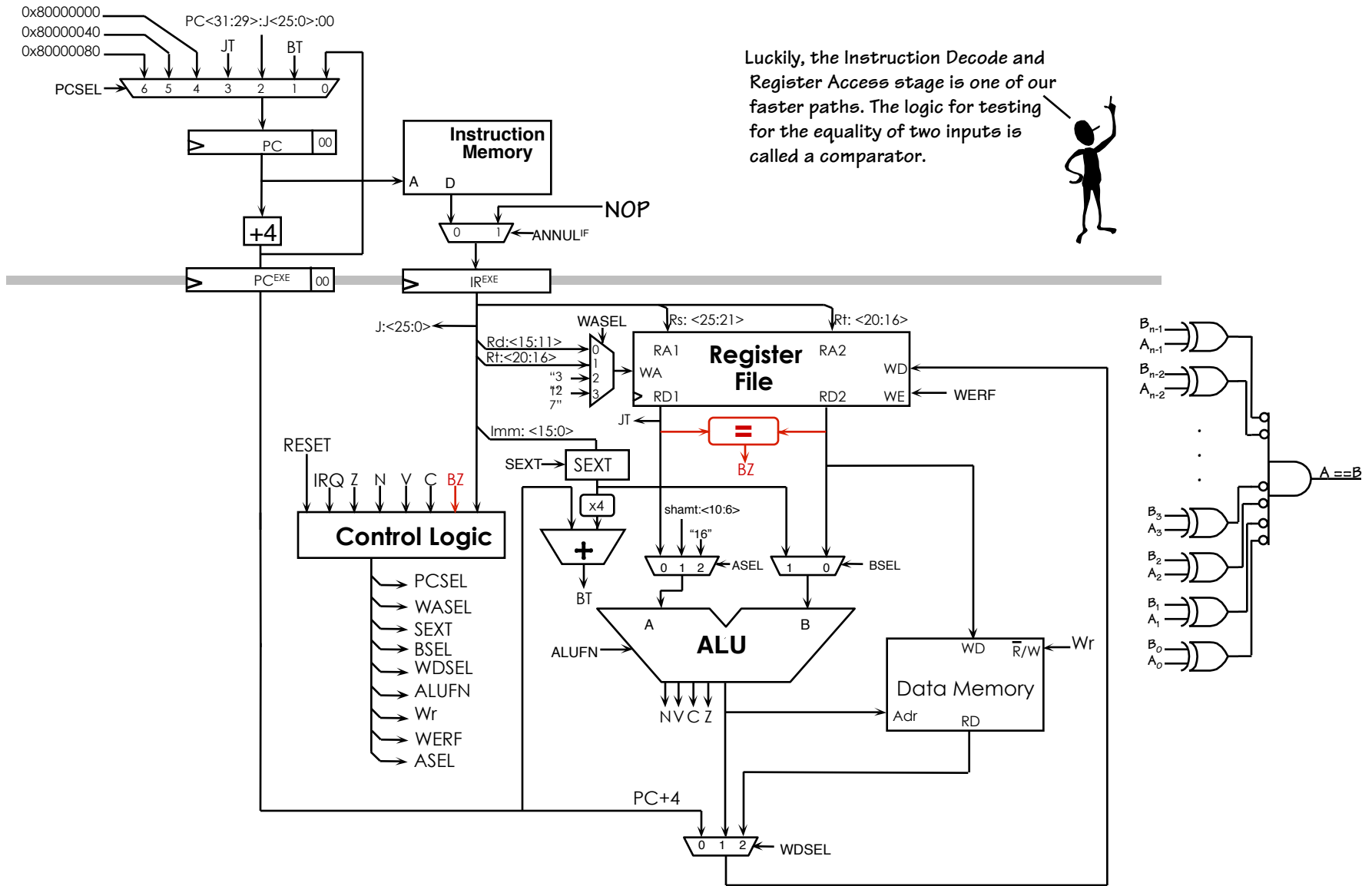
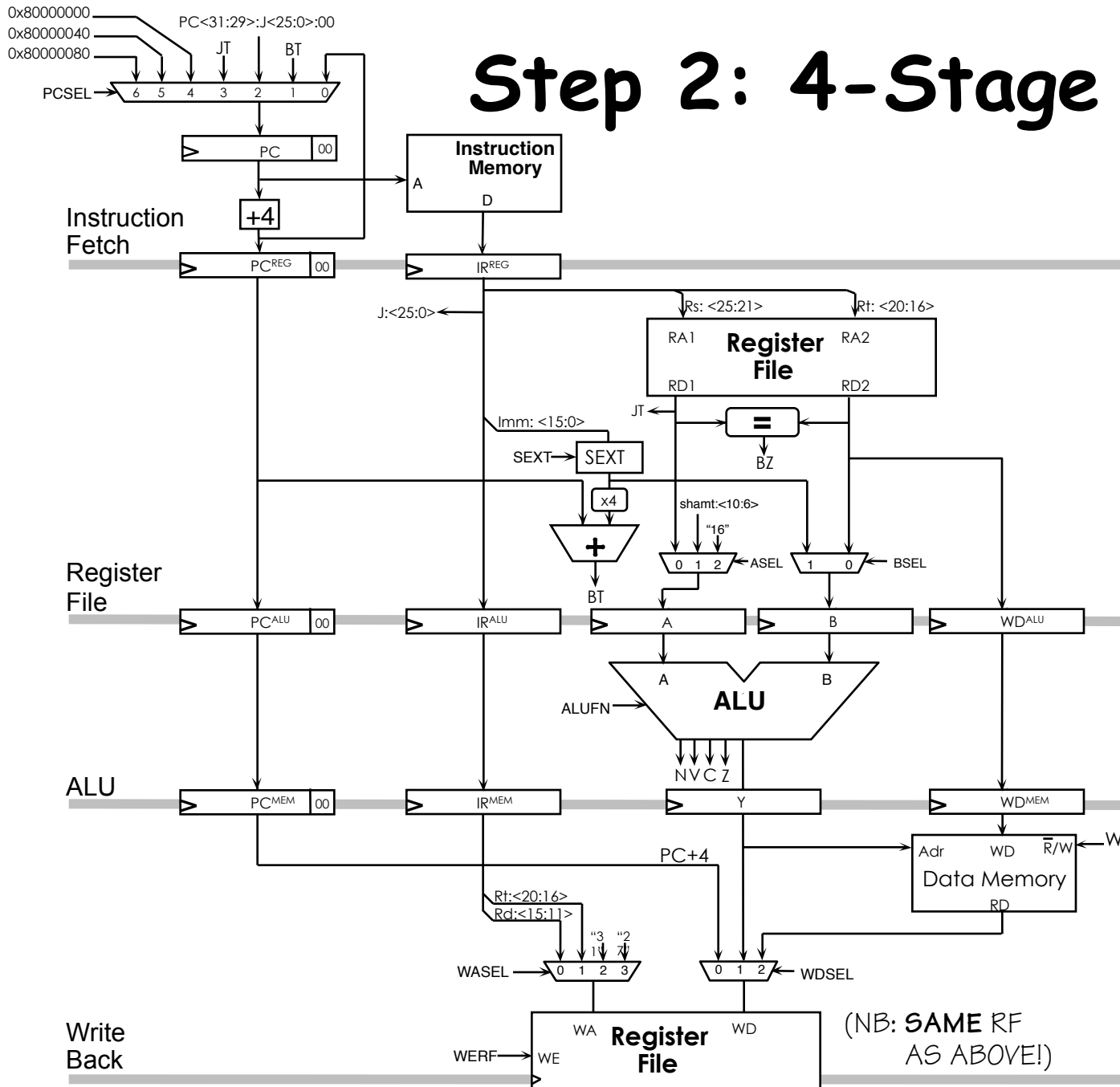Luckily, the Instruction Decode and Register Access stage is one of our faster paths. The logic for testing for the equality of two inputs is called a comparator.

0x80000000
0x80000040
0x80000080

PC<31:29>:J<25:0>:00

JT   BT

PCSEL→ 6 5 4 3 2 1 0

PC 00

Instruction Memory

A

D

Instruction Fetch

+4

PC^REG 00

IR^REG

J:<25:0>←

Rs: <25:21>   Rt: <20:16>

RA1   **Register File**   RA2

RD1   RD2

Imm: <15:0>

SEXT→ SEXT

x4

JT←   =

BZ

shamt:<10:6>

"16"

0 1 2 ←ASEL   1 0 ←BSEL

Register File

PC^ALU 00   IR^ALU   A   B   WD^ALU

BT

A   B

ALU

ALUFN→

N V C Z

ALU

PC^MEM 00   IR^MEM   Y   WD^MEM

PC+4

Adr   WD   R̄/W ←Wr

Data Memory

RD

Rt:<20:16>
Rd:<15:11>

"3 "2
1   0

WASEL→ 0 1 2 3   0 1 2 ←WDSEL

Write Back

WA **Register File** WD

WERF→ WE

(NB: **SAME** RF AS ABOVE!)

Treats register file as two separate devices: combinational READ, clocked WRITE at end of pipe.

What other information do we have to pass down pipeline?
   PC
   (return addresses)

   instruction fields
   (decoding)

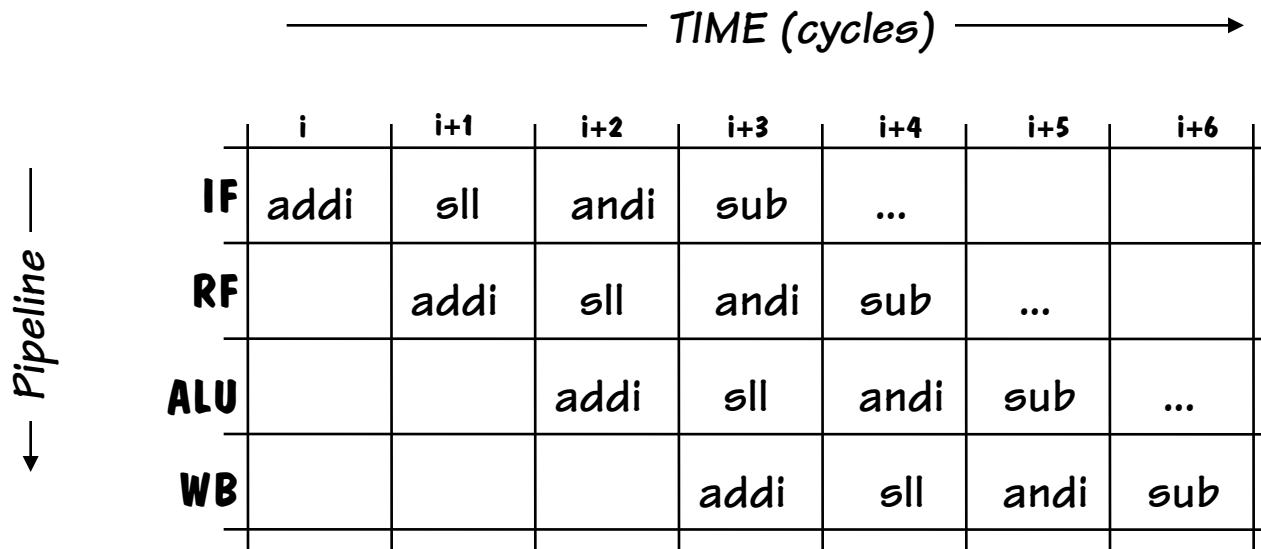What sort of improvement should expect in cycle time?

4-Stage miniMIPS Operation

Consider a sequence
of instructions:

```
...
addi $t0,$t0,1
sll  $t1,$t1,2
andi $t2,$t2,15
sub  $t3,$0,$t3
...
```

Executed on our 4-stage pipeline:

$\longrightarrow$ TIME (cycles) $\longrightarrow$

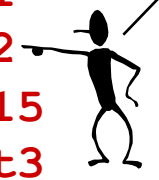|	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	sll	andi	sub	...		
RF		addi	sll	andi	sub	...	
ALU			addi	sll	andi	sub	...
WB				addi	sll	andi	sub

Pipeline

Pipeline "Structural Hazard"

BUT consider instead:

```
...
addi  $t0,$t0,1
sll   $t1,$t0,2
andi  $t2,$t2,15
sub   $t3,$0,$t3
...
```

One of our source operands is the destination of the previous instruction

Stuff like this never happened when we did pipelining before. Why now?

Before, we forbade feedback. Can't do that with a useful CPU.

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	sll	andi	sub			
RF		addi	sll	andi	sub		
ALU			addi	sll	andi	sub	
WB				addi	sll	andi	sub

How do we fix this one?

Oops! sll is trying to read Reg[8] ($t0) during cycle I+2 but addi doesn't write its result into Reg[8] until the end of cycle I+3!

Data Hazard Solution 1

"Program around it"

... *document weirdo semantics, declare it a software problem.*

       - Breaks sequential semantics!
       (Order of instruction execution is not obvious)
       - Costs code efficiency.

EXAMPLE: Rewrite

```
addi  $t0,$t0,1              addi  $t0,$t0,1
sll   $t1,$t0,2     as       andi  $t2,$t2,15
andi  $t2,$t2,15             sub   $t3,$0,$t3
sub   $t3,$0,$t3             sll   $t1,$t0,2
```

How often can we do this?

Not Very.

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	andi	sub	sll			
RF		addi	andi	sub	sll		
ALU			addi	andi	sub	sll	
WB				addi	andi	sub	sll

Data Hazard Solution 2

Stall the pipeline

    (add bubbles/disable update to IR^Xs and PC^Xs):

    Freeze IF, RF stages for 2 cycles, inserting NOPs

    into ALU-stage instruction register

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	sll	andi	andi	andi	sub	
RF		addi	sll	sll	sll	andi	sub
ALU			addi	NOP	NOP	sll	andi
WB				addi	NOP	NOP	sll

Drawback: Added NOPs "waste" cycles. Lot's of wasted cycles. (A large percentage of instructions depend on results from the immediately preceding instruction)

# Data Hazard Solution 3

Bypass (aka forwarding) Paths:

Add extra data paths & control logic to re-route data in problem cases.

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	sll	andi	sub			
RF		addi	sll	andi	sub		
ALU			addi	sll	andi	sub	
WB				addi	sll	andi	sub

Idea: The result from the `addi`, which will be written into the register file at the end of cycle I+3, *is actually available at output of the ALU during cycle I+2 — just in time for it to be input into the ALU of the `sll` in the RF stage! Thus, using it before it is actually written into the register!*
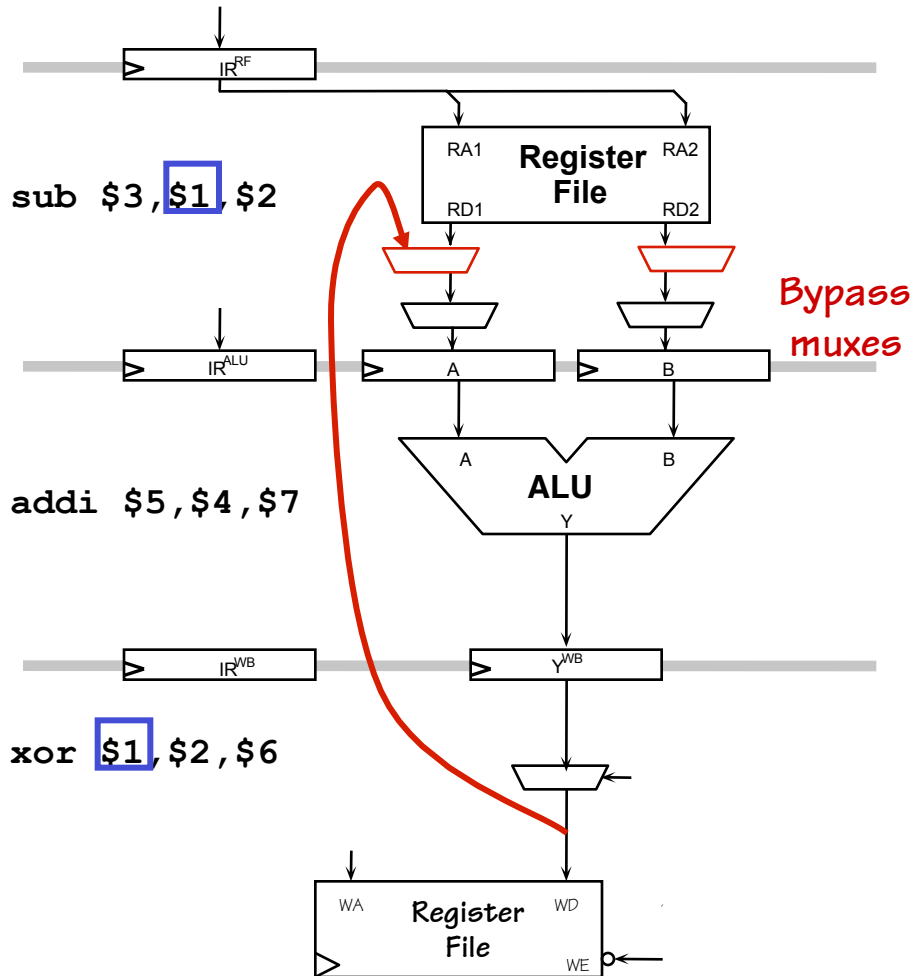
Bypass Paths (I)

sll $9,$8,2

addi $8,$8,1
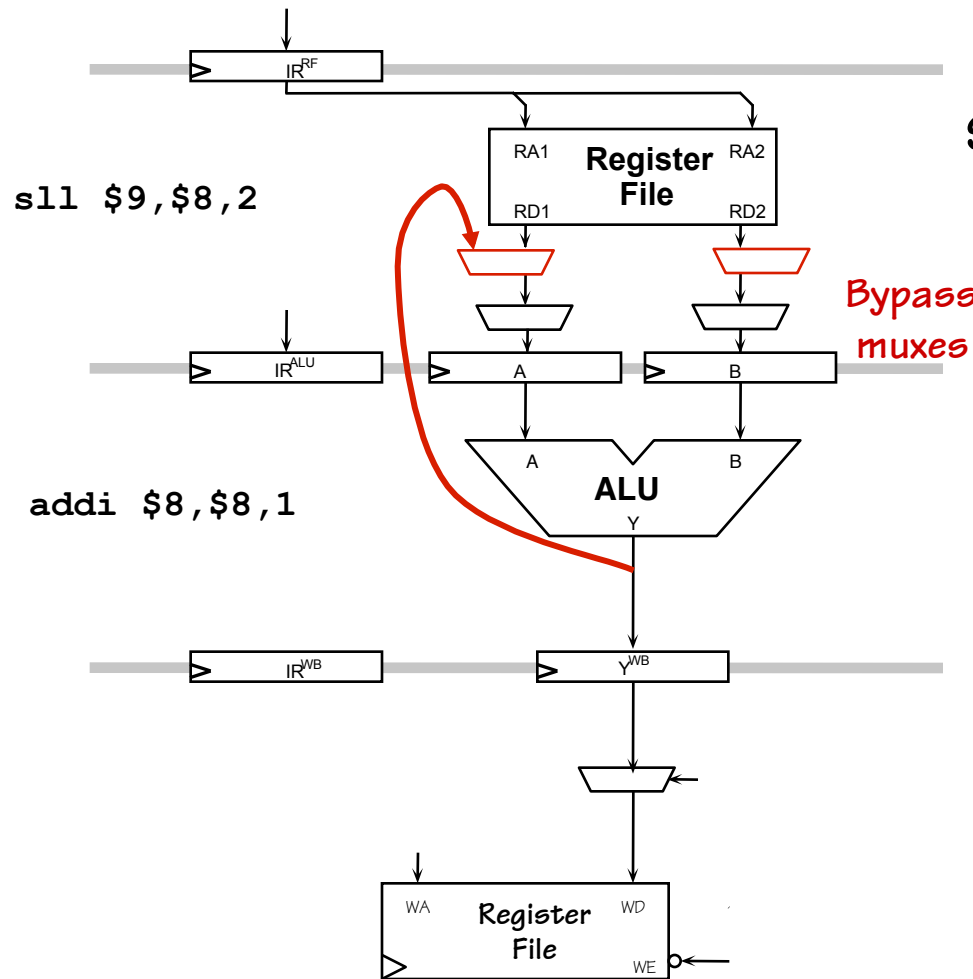
SELECT *this* BYPASS path if

Op^{RF} = reads Rs and
((Op^{ALU} = R-type and Rs^{RF} = Rd^{ALU})
                    or
(Op^{ALU} = I-type and Rs^{RF} = Rt^{ALU}))

i.e., instructions that update registers with ALU results

and Rs^{RF} != 0

Bypass Paths (II)

SELECT *this* BYPASS path if

Op^{RF} uses Rs^{RF} as a source
<u>and</u> Rs^{RF} != 0
<u>and</u> not using ALU bypass
<u>and</u> WERF = 1
<u>and</u> Rs^{RF} = WA

Why not get it from the register file? It's being written this cycle!

Next Time

Many More
Bypasses Ahead