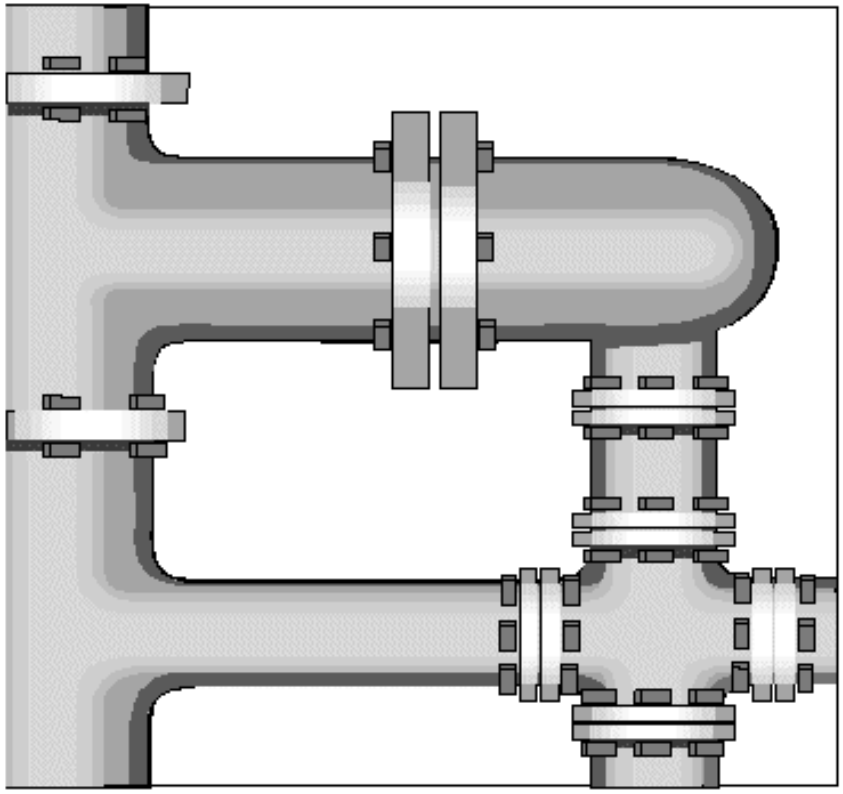
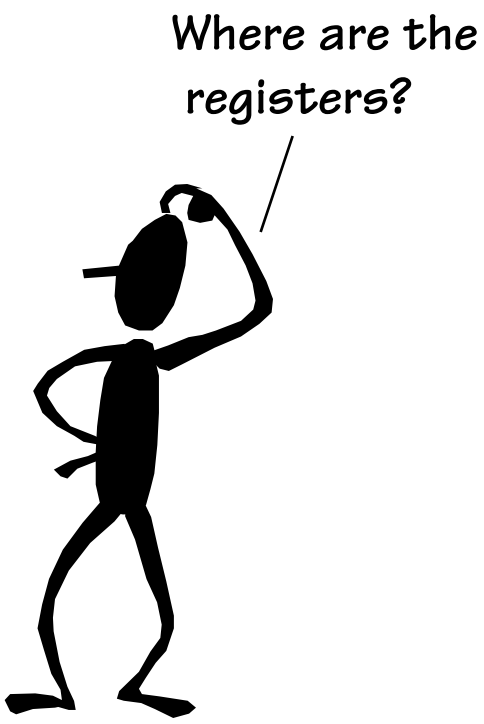
Pipelined CPUs





This week's lab will be held in Chapman 125

Review of CPU Performance

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

MIPS = Millions of Instructions/Second

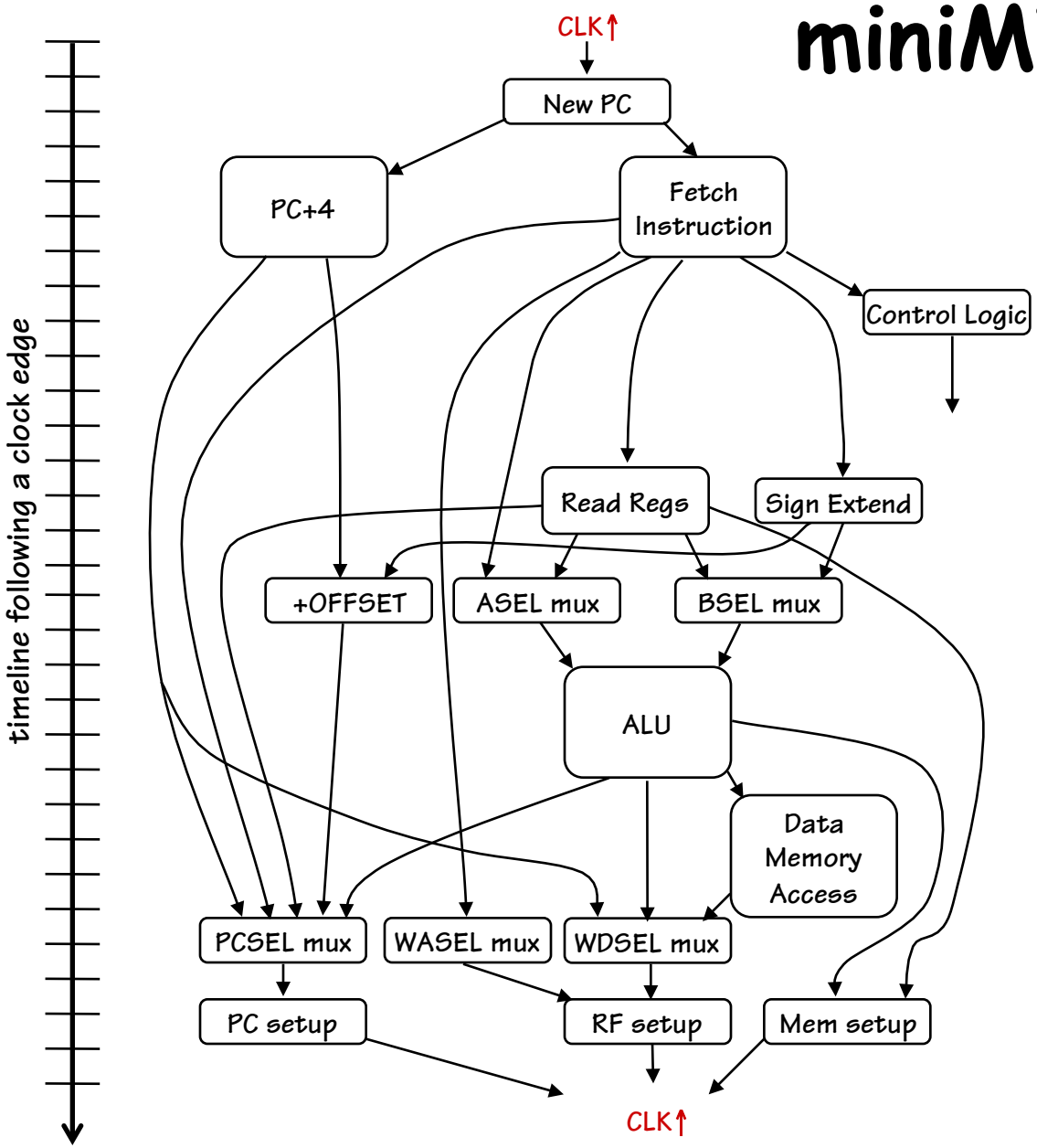
Freq = Clock Frequency, MHz

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



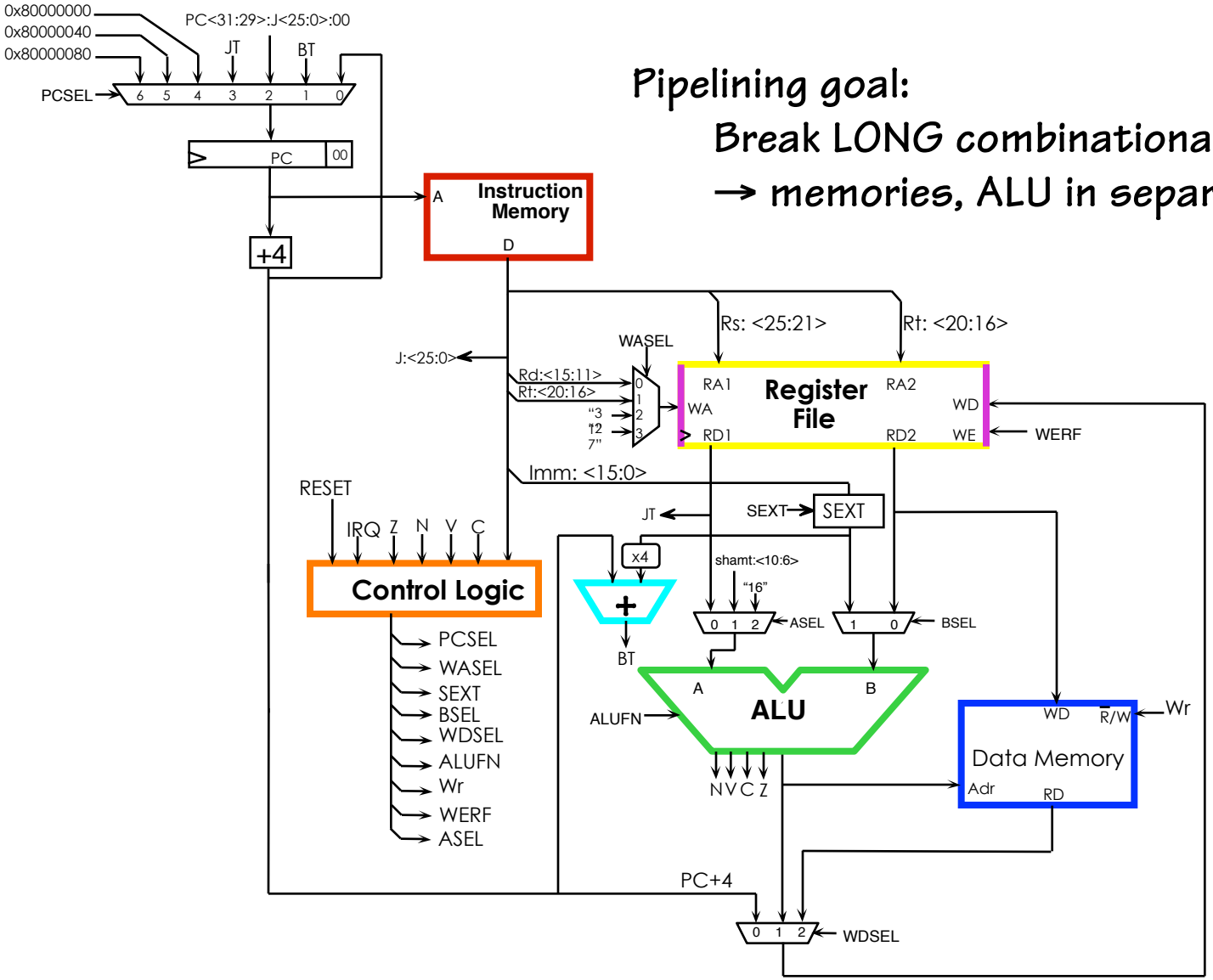
The diagram on the left illustrates the Data Flow through time of miniMIPS

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:

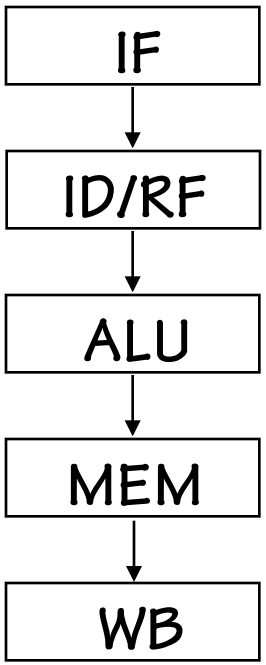
Break LONG combinational paths

→ 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:



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

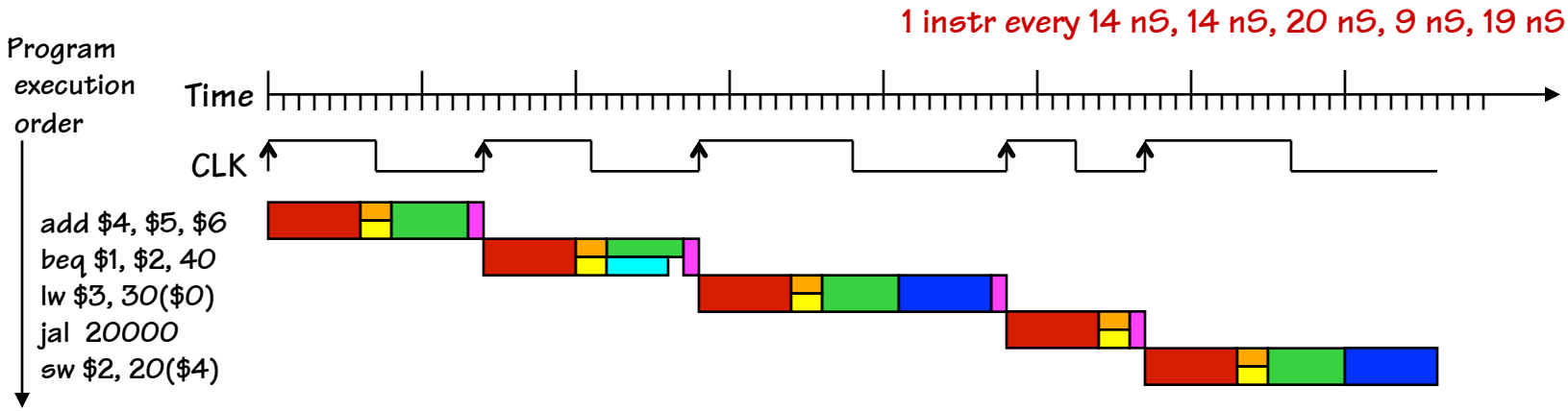
ALU stage: Performs specified operation, passes result to

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.

Current miniMIPS Timing

Different instructions use various parts of the data path.

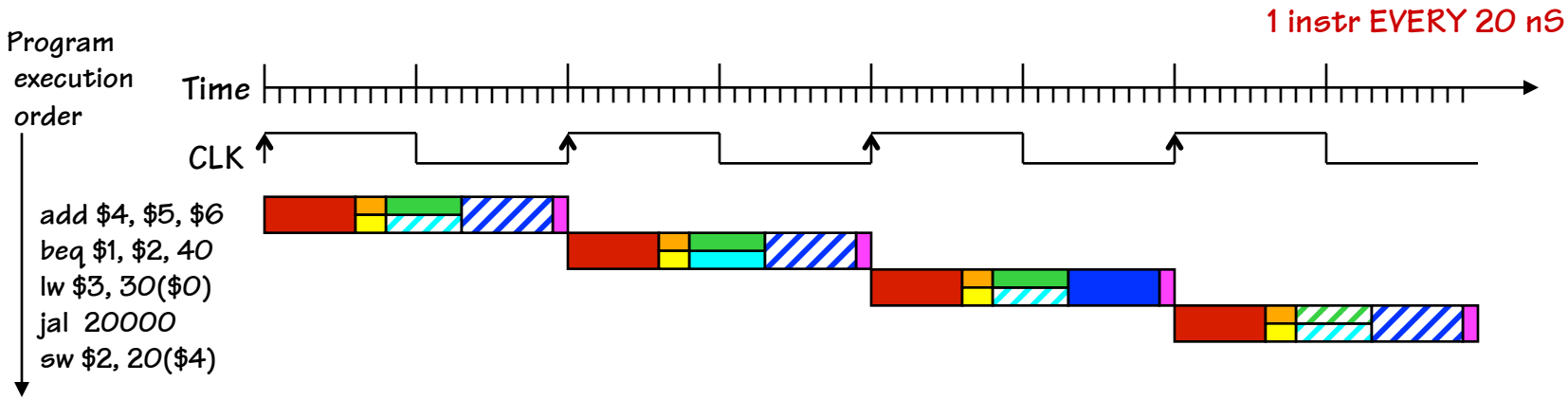


6 nS	Instruction Fetch
2 nS	Instruction Decode
2 nS	Register Prop Delay
5 nS	ALU Operation
4 nS	Branch Target
6 nS	Data Access
1 nS	Register Setup

This is an example of a "Asynchronous Globally-Timed" control strategy (see Lecture 18). 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!

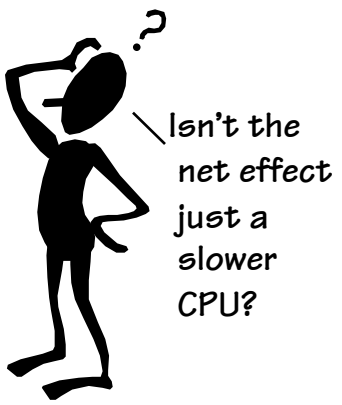
Uniform miniMIPS Timing

With a fixed clock period, we have to allow for the worse case.

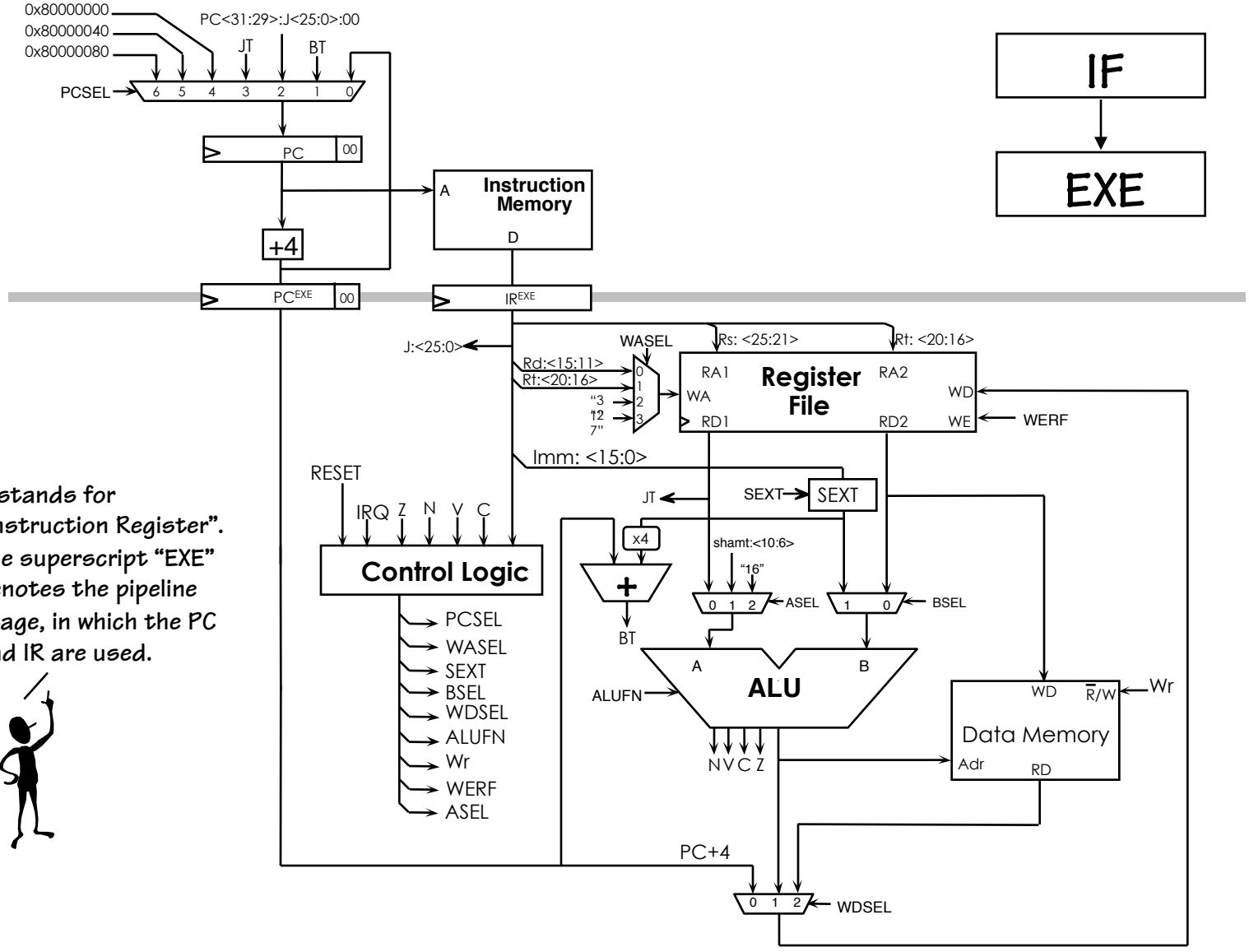


6 nS	Instruction Fetch
2 nS	Instruction Decode
2 nS	Register Prop Delay
5 nS	ALU Operation
4 nS	Branch Target
6 nS	Data Access
1 nS	Register Setup

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!



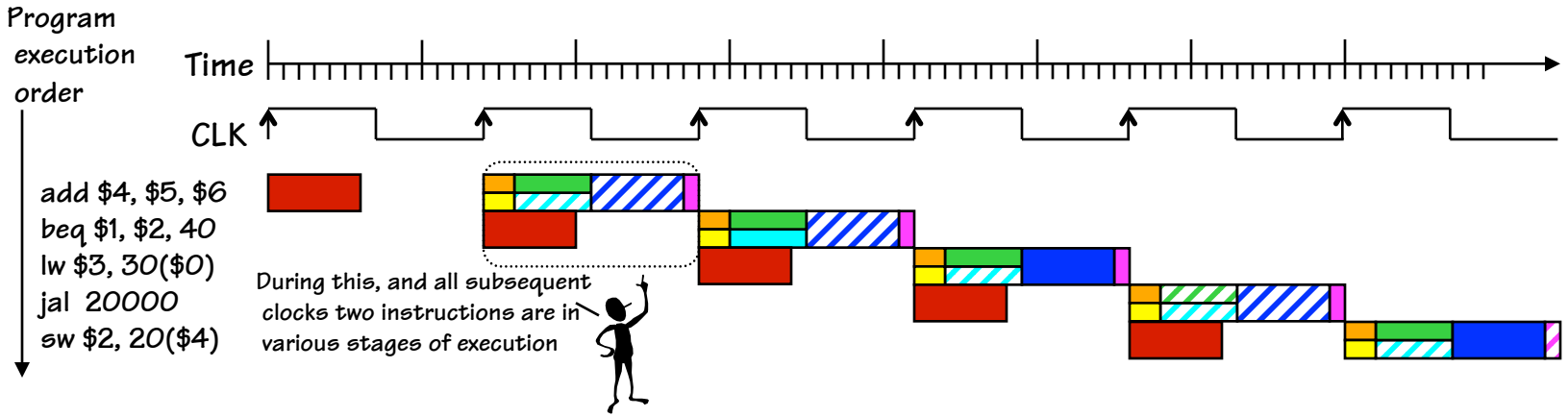
Step 1: A 2-Stage Pipeline



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.

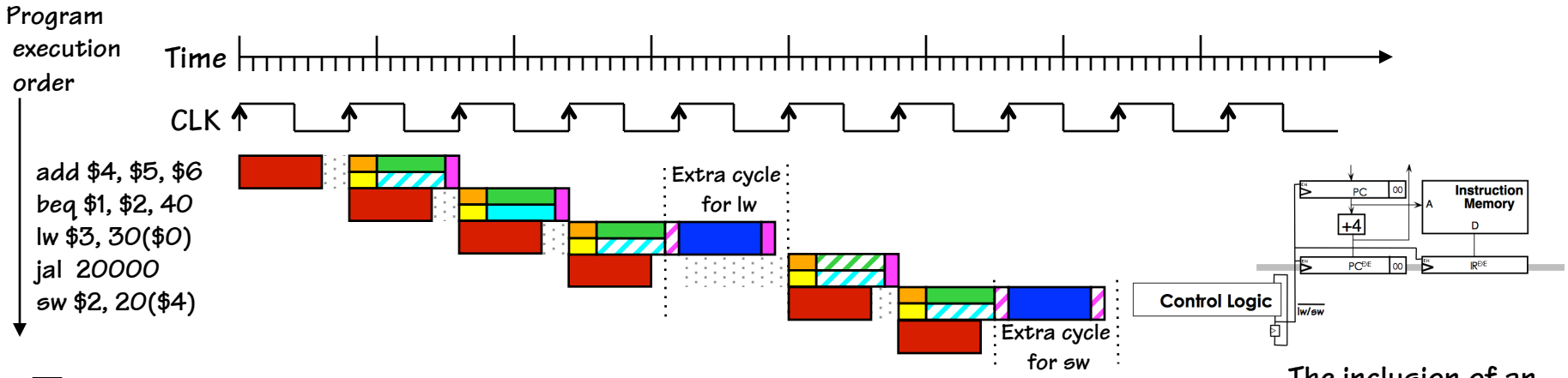


6 nS Instruction Fetch
2 nS Instruction Decode
2 nS Register Prop Delay
5 nS ALU Operation
4 nS Branch Target
6 nS Data Access
1 nS Register Setup

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.

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.



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 is now more than twice our original design (1/20nS). Does that mean it is twice as fast?

Not likely. In SPEC benchmarks, as many as 30% of instructions access memory. Thus, the effective speed up is:

$$speed\ up = \frac{old\ clock\ period}{new\ clock\ period\ (1*0.7+2*0.3)} = \frac{20}{8\,(1.3)} = 1.923$$

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

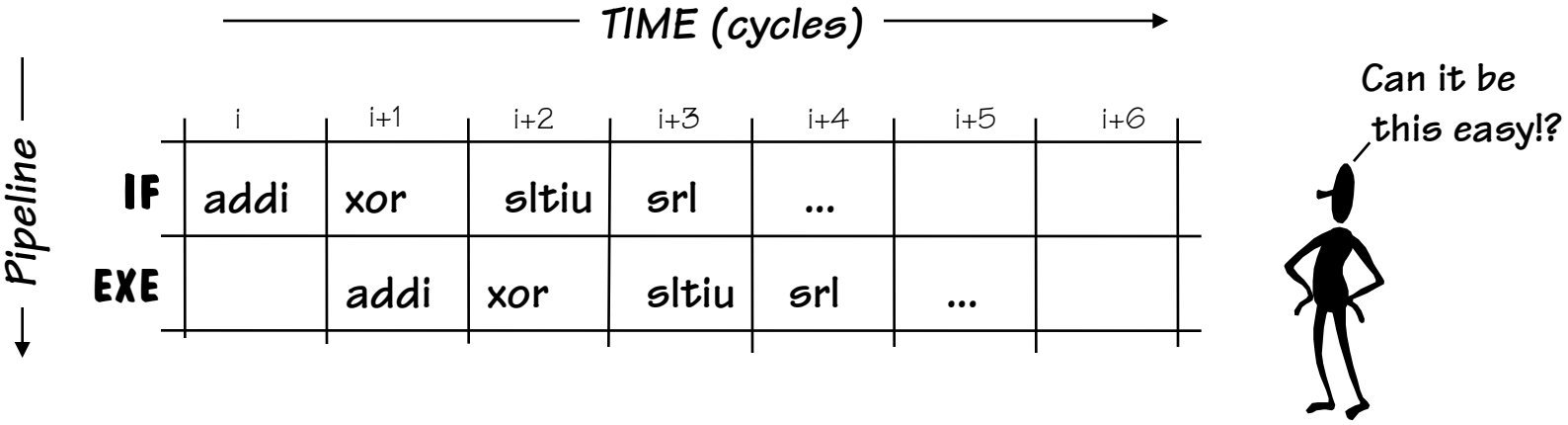
2-Stage Pipelined Operation

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 18.

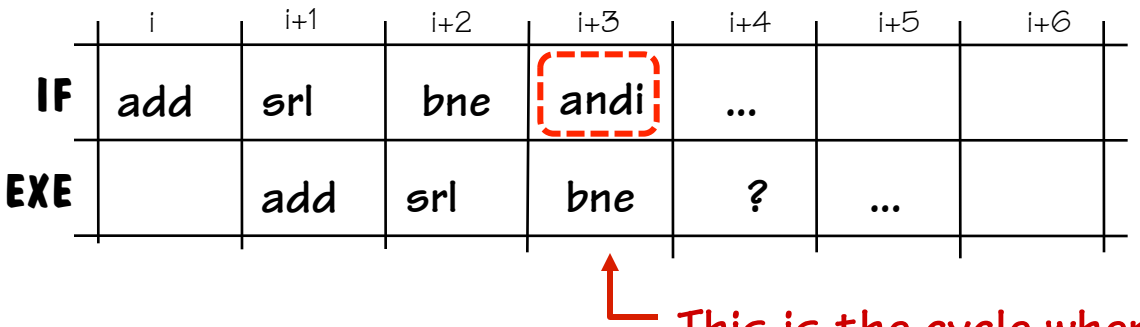
Executed on our 2-stage pipeline:



Pipeline Control Hazards

BUT consider instead:

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



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.

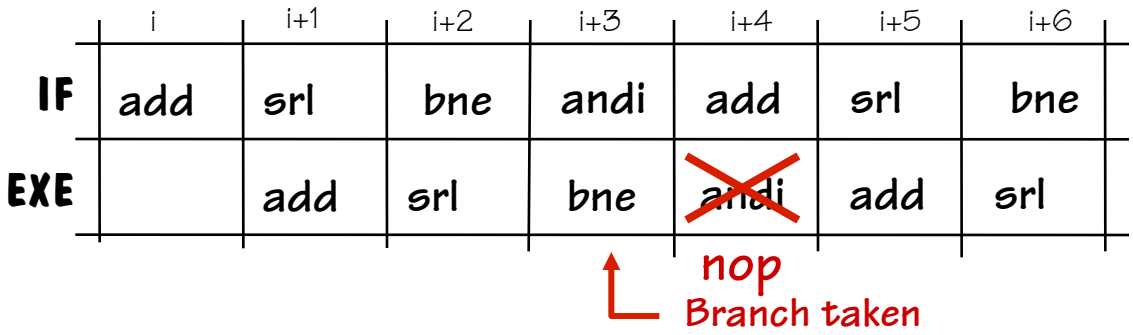
2. "Program around it". Either
   a) Follow each BNE/BEQ with a NOP instruction; or
   b) Make compiler clever enough to move USEFUL instructions into the branch delay slots
      i. Always execute instructions in delay slots
      ii. Conditionally execute instructions in delay slots

- 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.

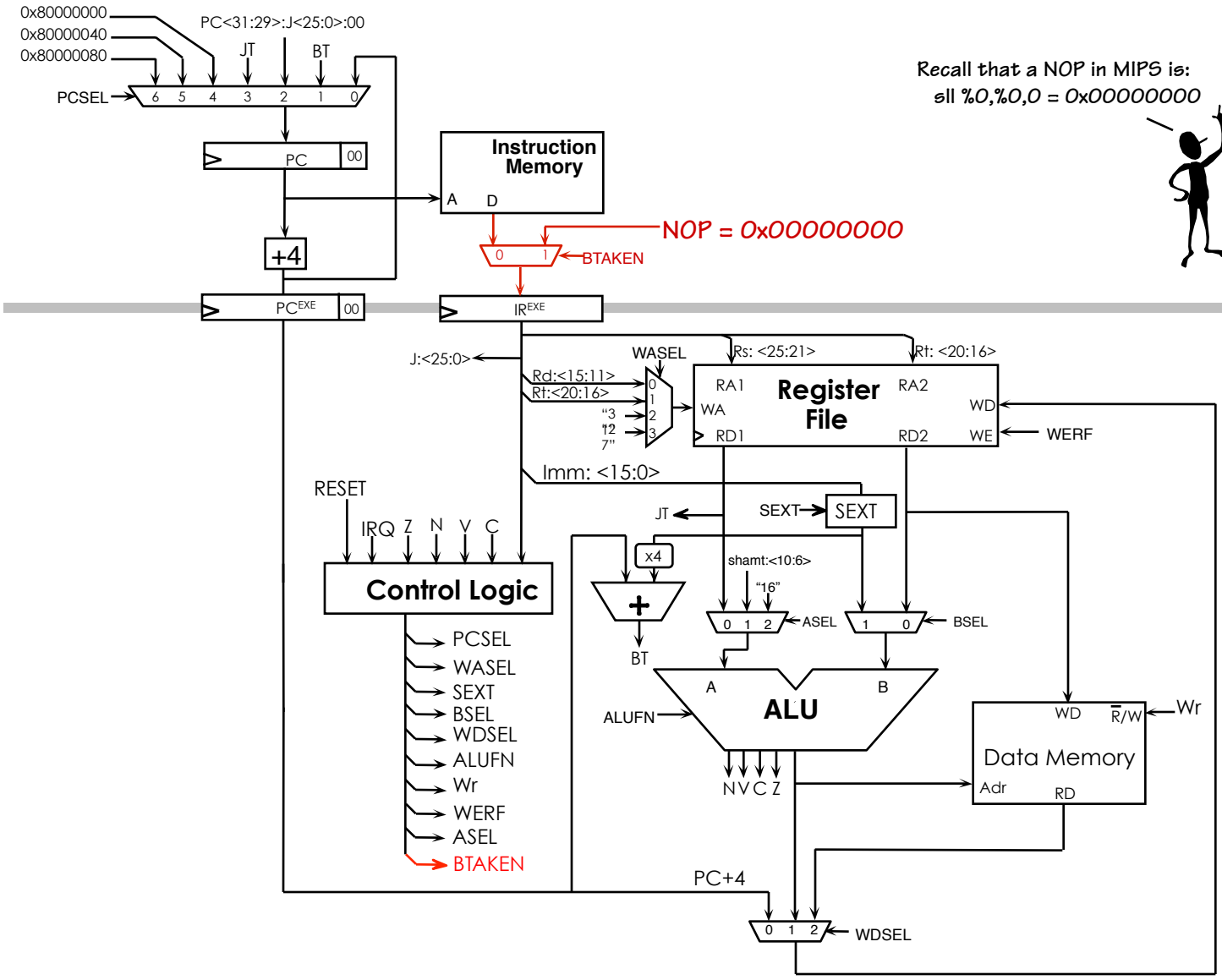
```
loop:  add  $t1,$t1,$t0
       srl  $t2,$t2,1
       bne  $t2,$0,loop
       andi $t0,$t2,1
       ...
```



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

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
       ...
```

	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)

This breaks the "SEQUENTIAL SEMANTICS" of the ISA. Logically, the branch takes place after the instruction in the DELAY SLOT is executed.

	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.

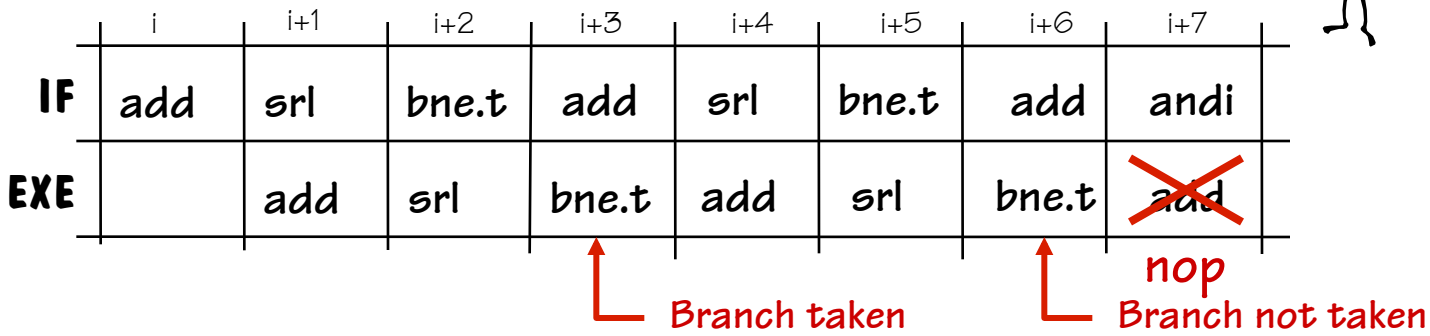
This is the standard approach for pipelined MIPS implementations

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.



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

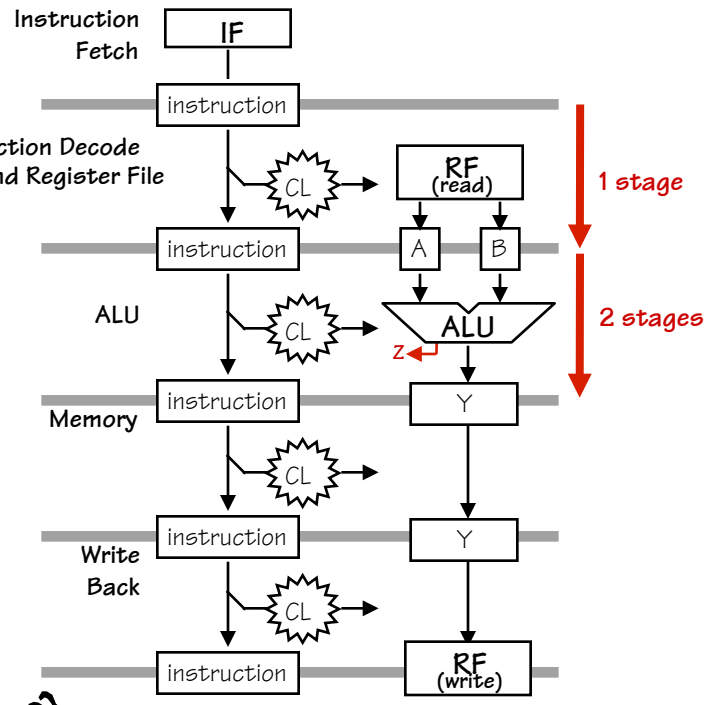
How is the number of branch delay slots determined? Depends on "where" in the pipeline the "branch decision" is made relative to where instructions are fetched.

Consider the 5-stage miniMIPS pipeline shown on the right.

What stage is the branch decision made?

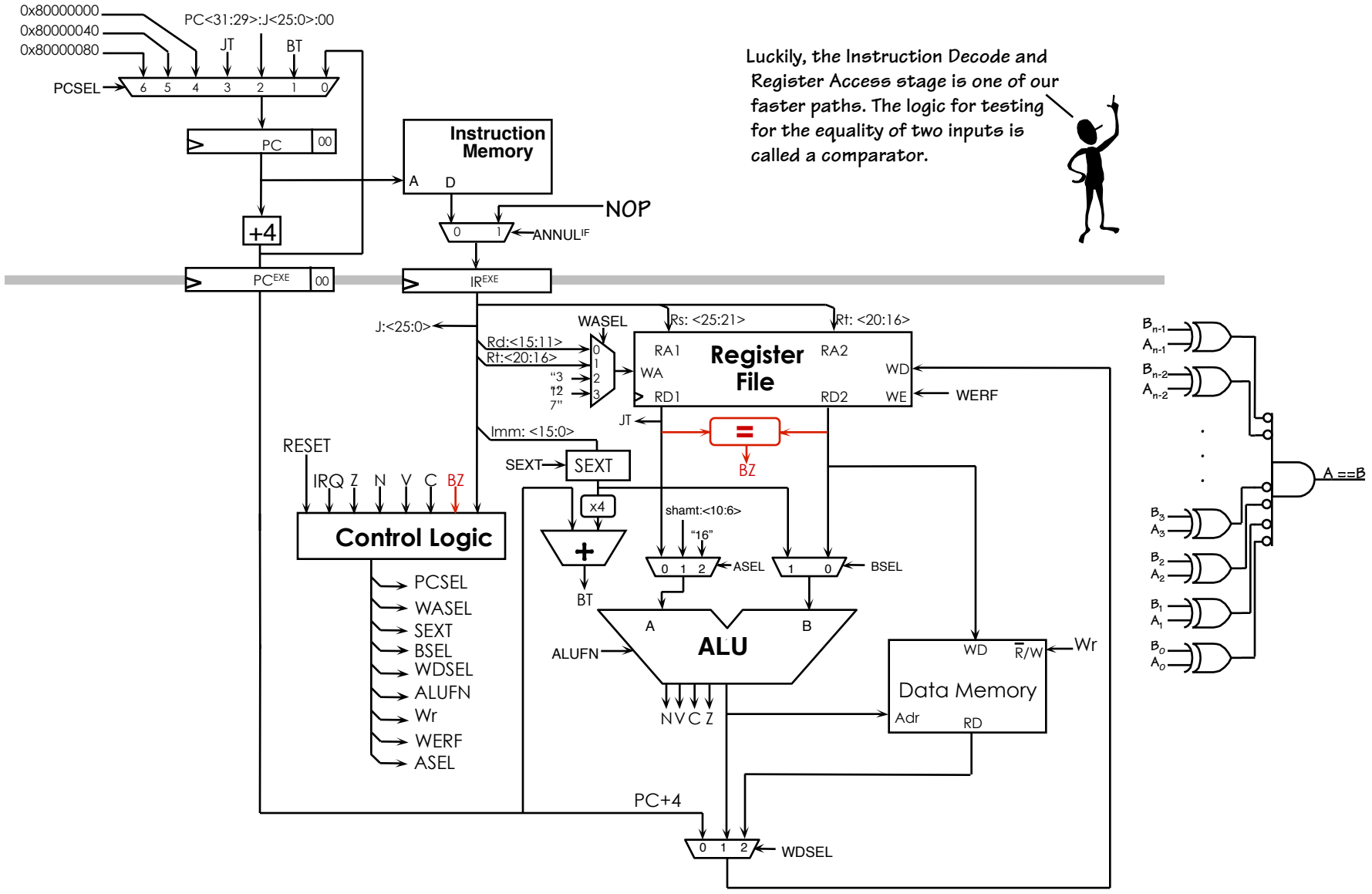
```
beq rs,rt,offset
   if (Reg[rs] == Reg[rt])
      PC ← PC + 4 + 4*SEXT(offset)
```

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



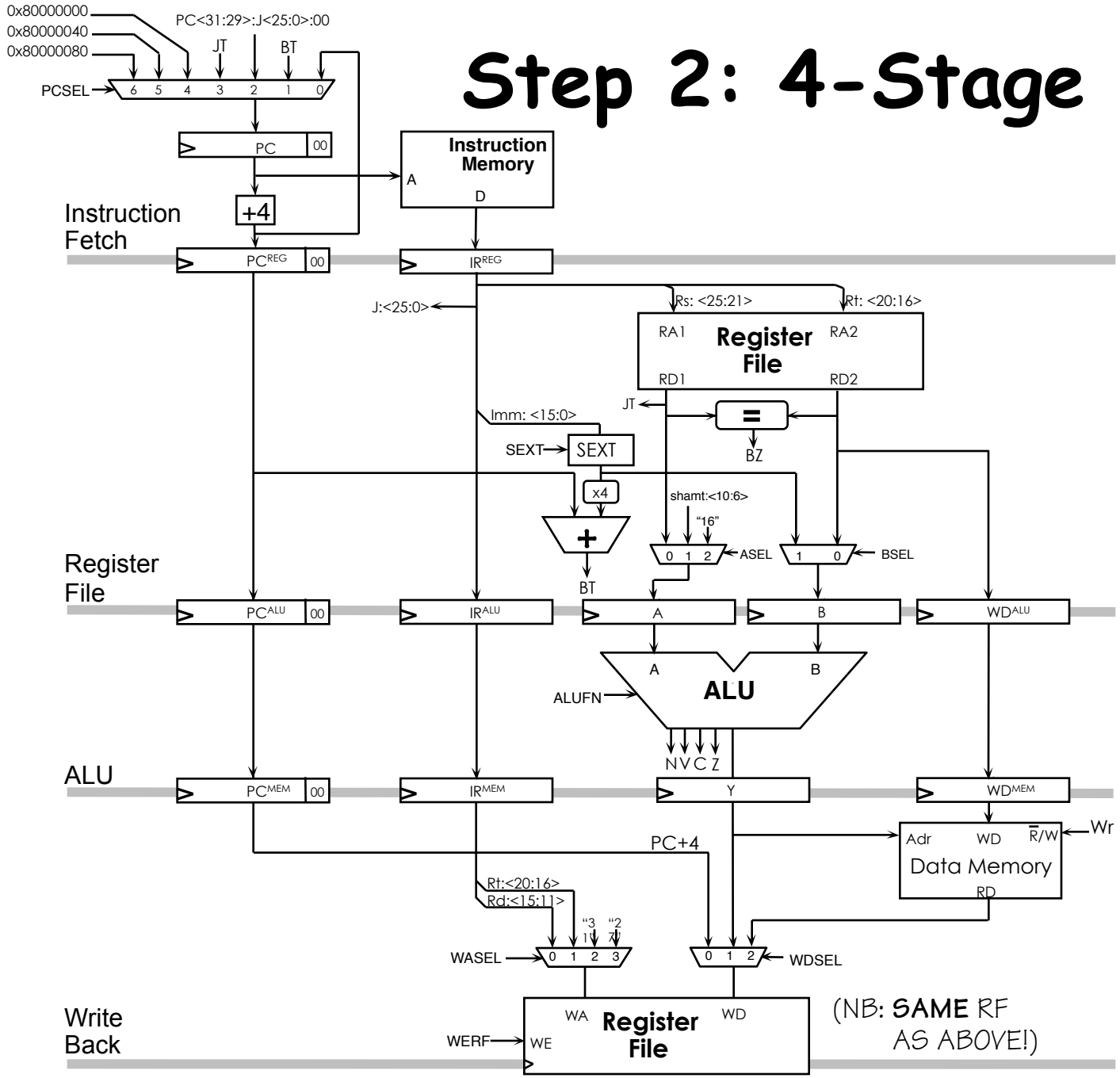
Is there any way miniMIPS' could make its branch decision sooner? We only need to support BNE and BEQ, since we choose to trap and emulate the more complicated branch instructions.

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.

Step 2: 4-Stage miniMIPS



Treats register file as two separate devices: combinational READ, and a synchronous WRITE at end of clock cycle.

What other information do we have to pass down pipeline?

- PC
- (return addresses)
- instruction fields
- (decoding)

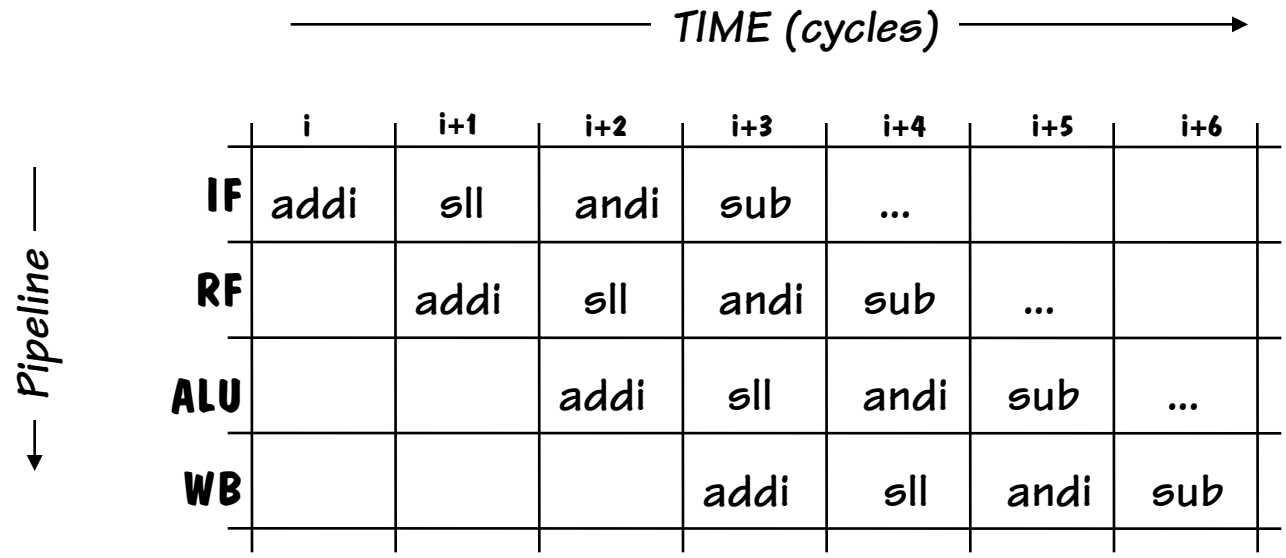
What sort of improvement should expect in cycle time?

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:

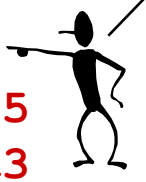


Pipeline "Structural Hazard"

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



	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

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!

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

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

How do we fix this one?

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

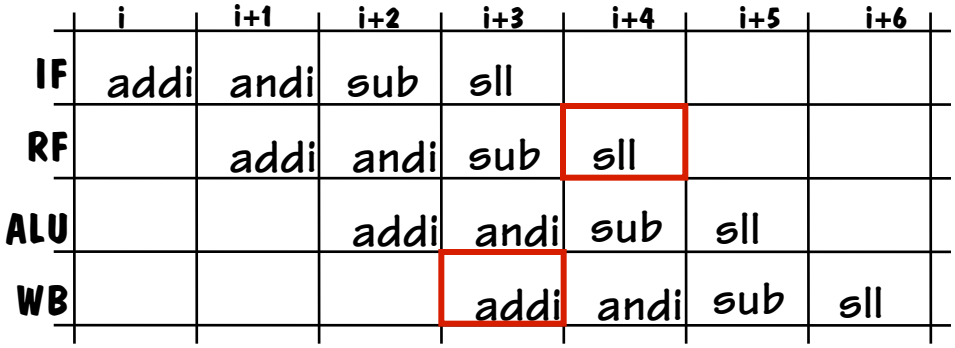
```
addi $t0,$t0,1
sll  $t1,$t0,2
andi $t2,$t2,15
sub  $t3,$0,$t3
```

as

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

How often can we do this?

Not Very.



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

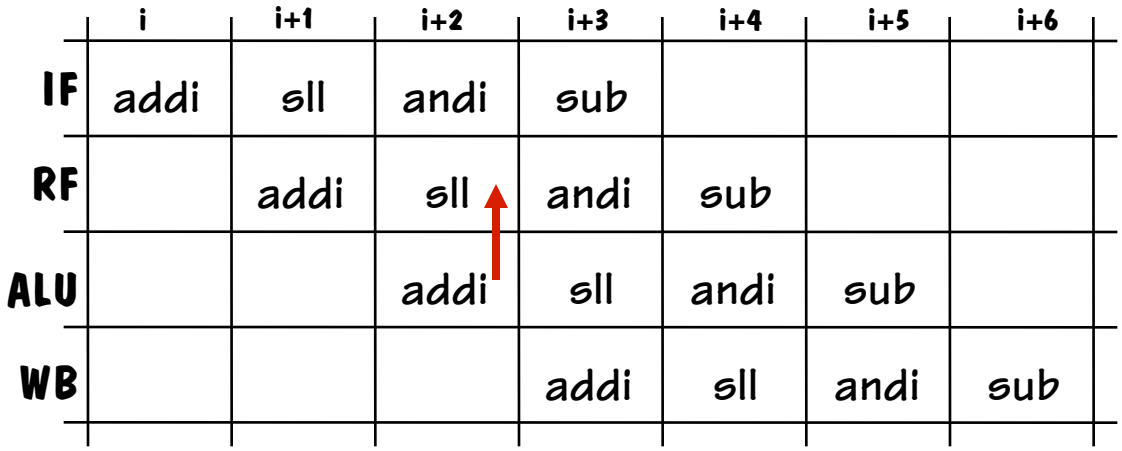
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	sll	andi	andi	andi	sub	
RF		addi	sll	sll	sll	andi	sub
ALU			addi	NOP	NOP	sll	andi
WB				addi	NOP	NOP	sll

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

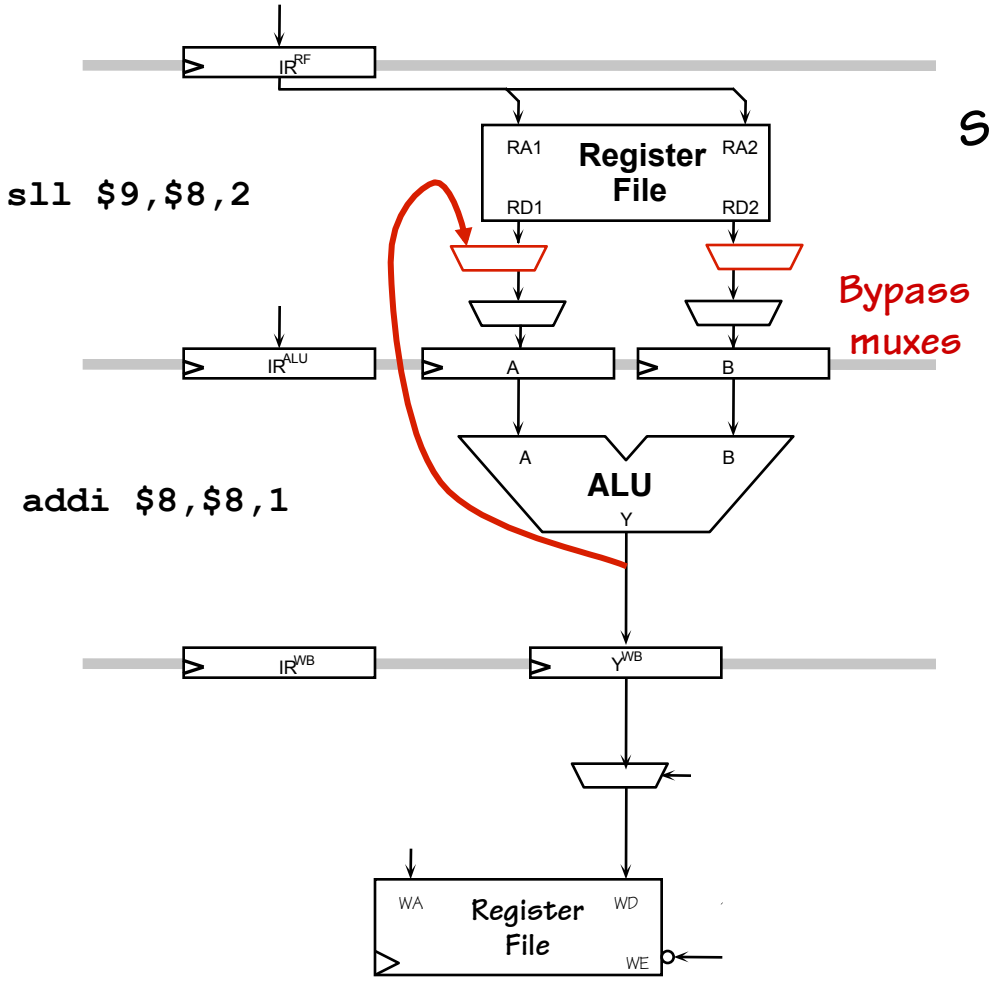
Bypass (aka forwarding) Paths:

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



IDEA: The ALU 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* near the end of 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)



SELECT *this* BYPASS path if

Op^{RF} = reads Rs and

((Op^{ALU} = R-type and Rs^{RF} = Rd^{ALU})

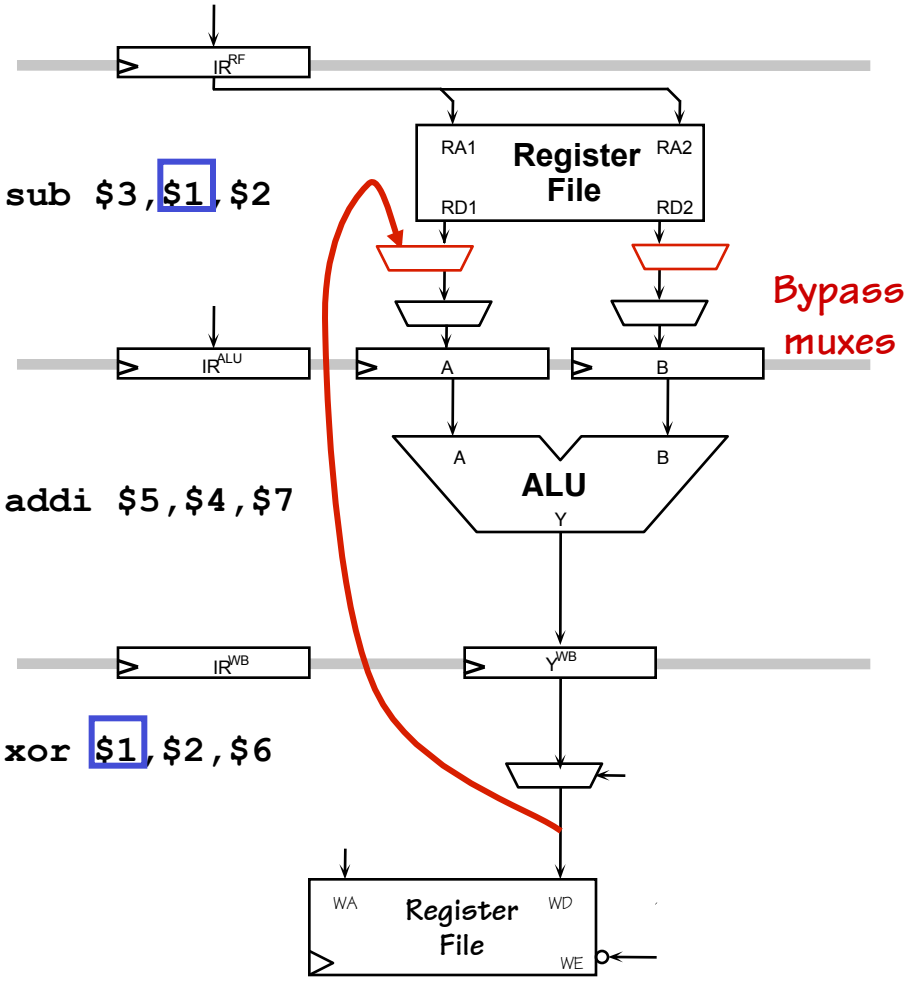
or

(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
and Rs^{RF} != 0
and not using ALU bypass
and WERF = 1
and Rs^{RF} = WA

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

Next Time