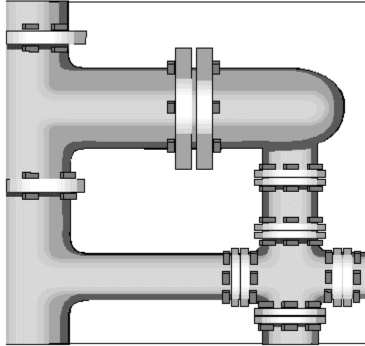
Pipelined CPUs



Study Chapter 6 of Text

Amdahl's Law

$$t_{improved} = \frac{t_{affected}}{r_{speedup}} + t_{unaffected}$$

Example:

"Suppose a program runs in 100 seconds on a machine, where multiplies are responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

25 = 80/r + 20 r = 16x

How about making it 5 times faster?

20 = 80/r + 20 r = ?

Principle: Make the common case fast

Example

Suppose we enhance a machine making all floating-point instructions run five times faster. If the execution time of some benchmark before the floating-point enhancement is 10 seconds, what will the speedup be if only half of the 10 seconds is spent executing floating-point instructions?

We are looking for a benchmark to show off the new floating-point unit described above, and want the overall benchmark to show a speedup of 3. One benchmark we are considering runs for 100 seconds with the old floating-point hardware. How much of the execution time would floating-point instructions have to account for in this program in order to yield our desired speedup on this benchmark?

Remember

Performance is specific to a particular programs

Total execution time is a consistent summary of performance

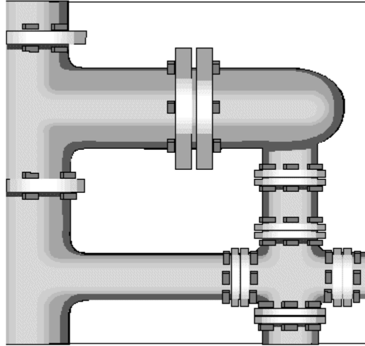
For a given architecture performance increases come from:

- increases in clock rate (without adverse CPI affects)
- improvements in processor organization that lower CPI
- compiler enhancements that lower CPI and/or instruction count

Pitfall: Expecting improvements in one aspect of a machine's performance to affect the total performance

You should not always believe everything you read! Read carefully!

Pipelined CPUs



Where are the registers?

Study Chapter 6 of Text

Review of CPU Performance

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

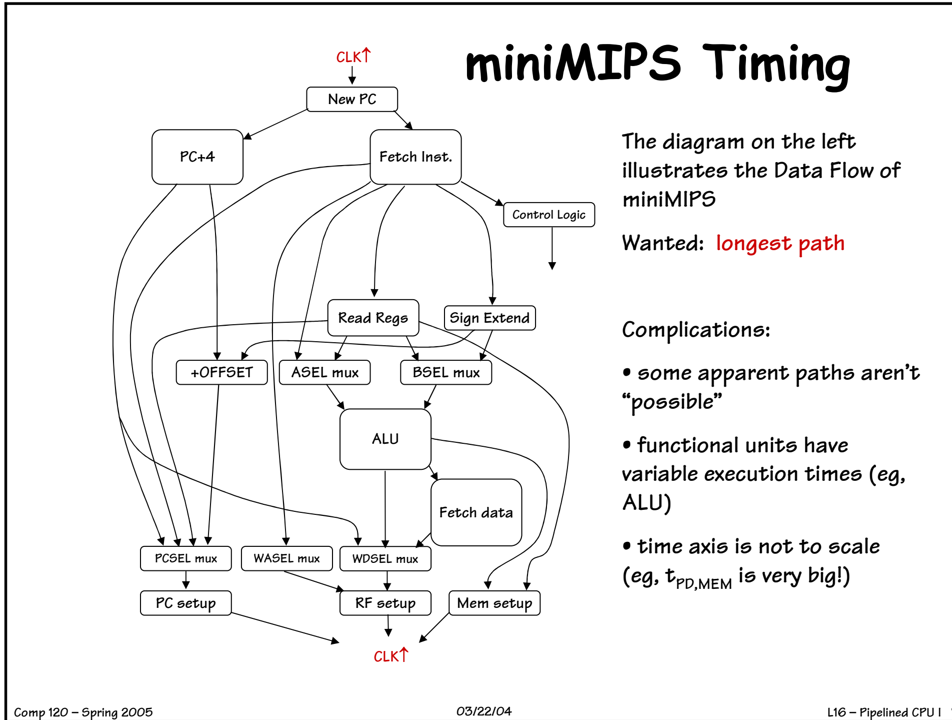
MIPS = Millions of Instructions/Second

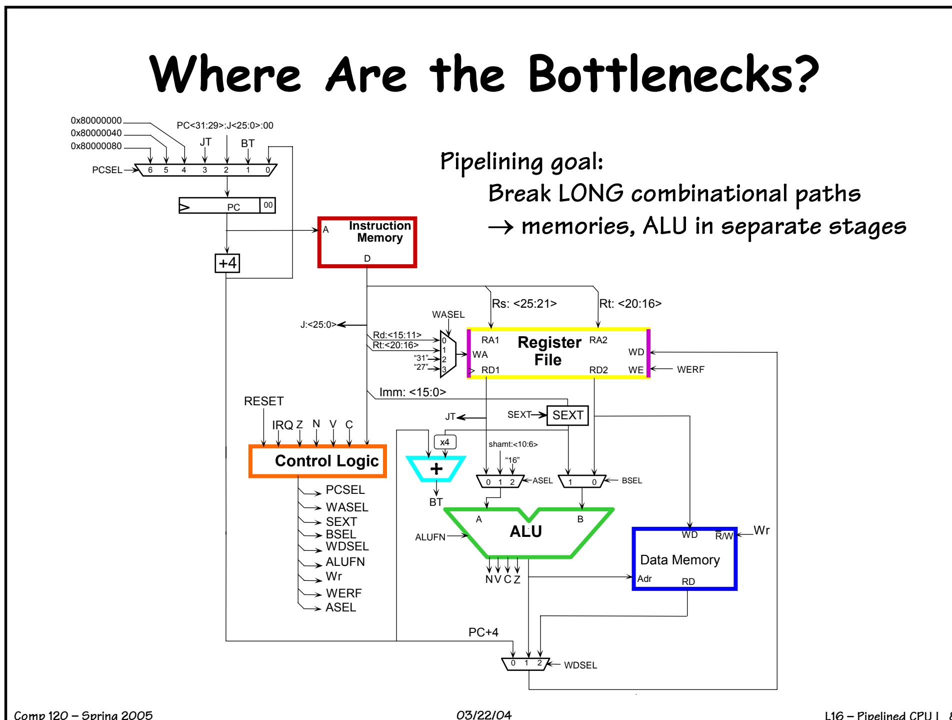
Freq = Clock Frequency, MHz

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.

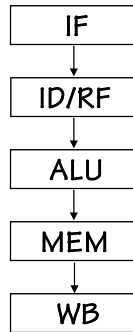

miniMIPS Timing
CLK↑
New PC
PC+4
Fetch Inst.
Control Logic
Read Regs
Sign Extend
+OFFSET
ASEL mux
BSEL mux
ALU
Fetch data
PCSEL mux
WASEL mux
WDSEL mux
PC setup
RF setup
Mem setup
CLK↑
The diagram on the left illustrates the Data Flow of miniMIPS
Wanted: longest path
Complications:
• some apparent paths aren't "possible"
• functional units have variable execution times (eg, ALU)
• time axis is not to scale (eg, $t_{PD,MEM}$ is very big!)
Comp 120 – Spring 2005
03/22/04
L16 – Pipelined CPU I 7


Where Are the Bottlenecks?
Pipelining goal:
Break LONG combinational paths
→ memories, ALU in separate stages
Instruction Memory
Register File
Control Logic
ALU
Data Memory
PCSEL
WASEL
SEXT
BSEL
WDSEL
ALUFN
Wr
WERF
ASEL
RESET
Rs: <25:21>
Rt: <20:16>
Imm: <15:0>
PC+4
Comp 120 – Spring 2005
03/22/04
L16 – Pipelined CPU I 8

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

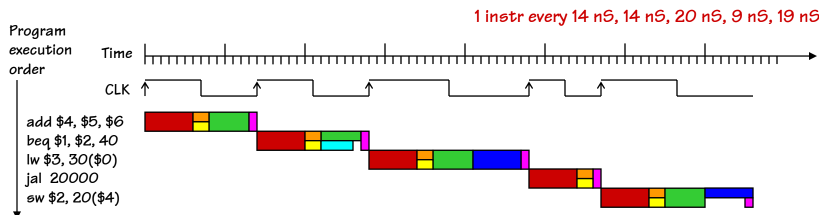
ALU stage: Performs specified operation, passes result to

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.

miniMIPS Timing

Different instructions use various parts of the data path.

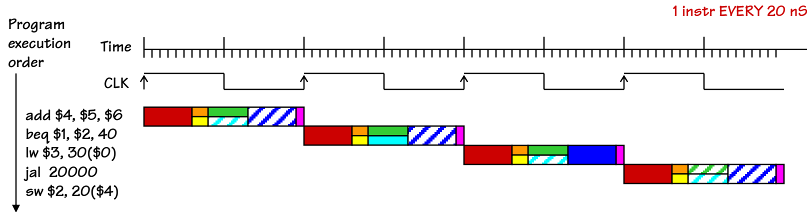


6 nS	Instruction Fetch
2 nS	Instruction Decode
2 nS	Register Prop Delay
5 nS	ALU Operation
4 nS	Branch Target
6 nS	Data Access
1 nS	Register Setup

This is an example of a "Asynchronous Globally-Timed" control strategy (see Lecture 8). 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!

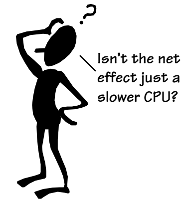
Uniform miniMIPS Timing

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

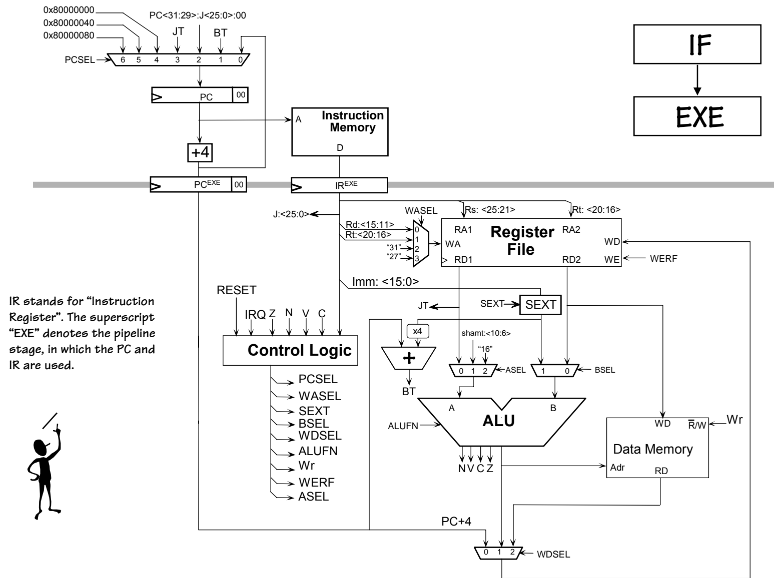


- Instruction Fetch
- Instruction Decode
- Register Prop Delay
- ALU Operation
- Branch Target
- Data Access
- Register Setup

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

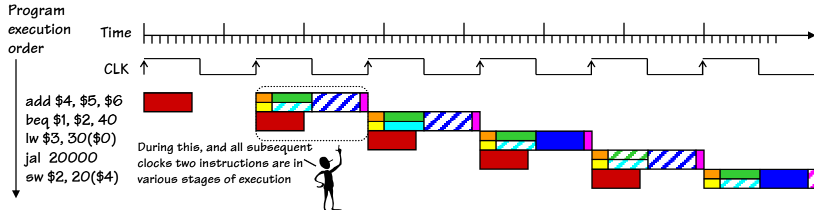


IF → EXE

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

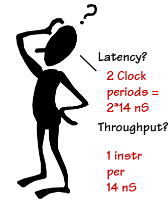
2-Stage Pipe Timing

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



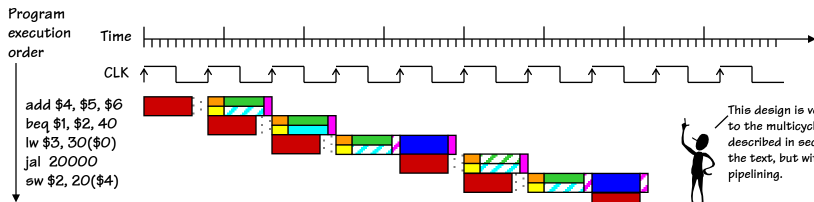
- Instruction Fetch
- Instruction Decode
- Register Prop Delay
- ALU Operation
- Branch Target
- Data Access
- Register Setup

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.



- Instruction Fetch
- Instruction Decode
- Register Prop Delay
- ALU Operation
- Branch Target
- Data Access
- Register Setup

The clock rate of this variant is more than 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:

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

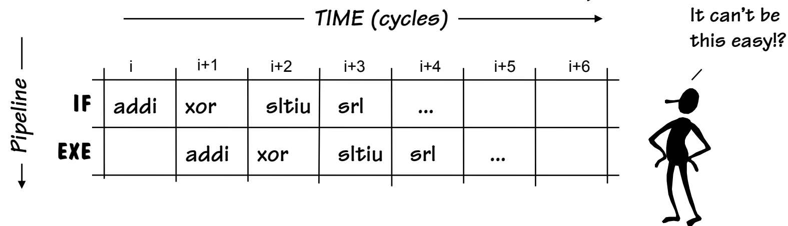
2-Stage Pipelined Operation

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 8.

Executed on our 2-stage pipeline:



TIME (cycles) →

Pipeline ↓

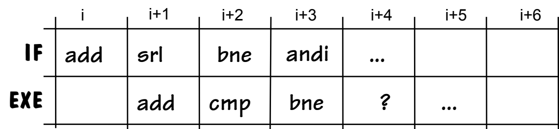
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	addi	xor	sltiu	srl	...		
EXE		addi	xor	sltiu	srl	...	

It can't be this easy!?

Pipeline Control Hazards

BUT consider instead:

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

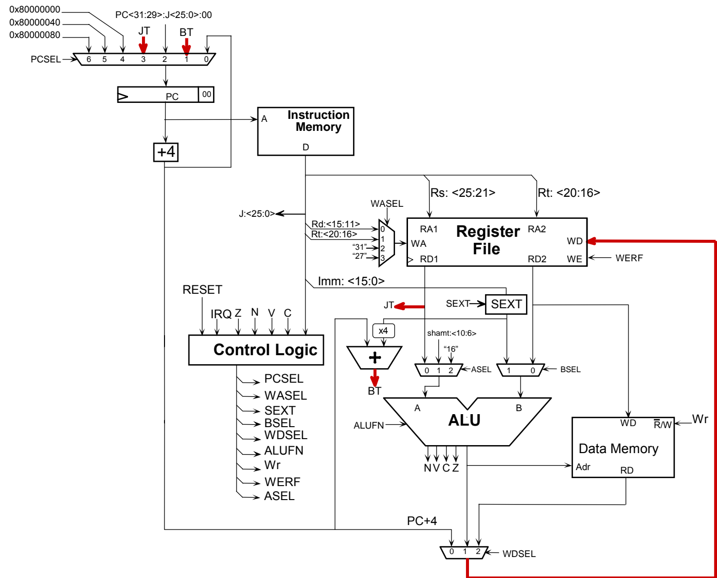


	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	add	srl	bne	andi	...		
EXE		add	cmp	bne	?	...	

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.

Feedback: potential hazard



Branch Delay Slots

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

POSSIBLE SOLUTIONS:

1. Make hardware "annul" 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

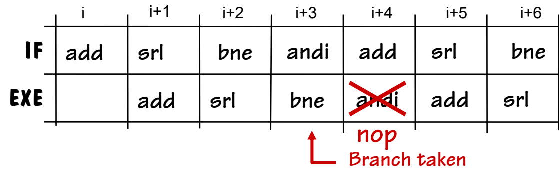
Solution 2 implies breaking the sequential semantics of the ISA. Logically, the branch takes place after instructions in the DELAY SLOTS are executed.

Delay slots also apply to the jump instructions, `j`, `jal`, and `jr`

Branch Solution 1

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

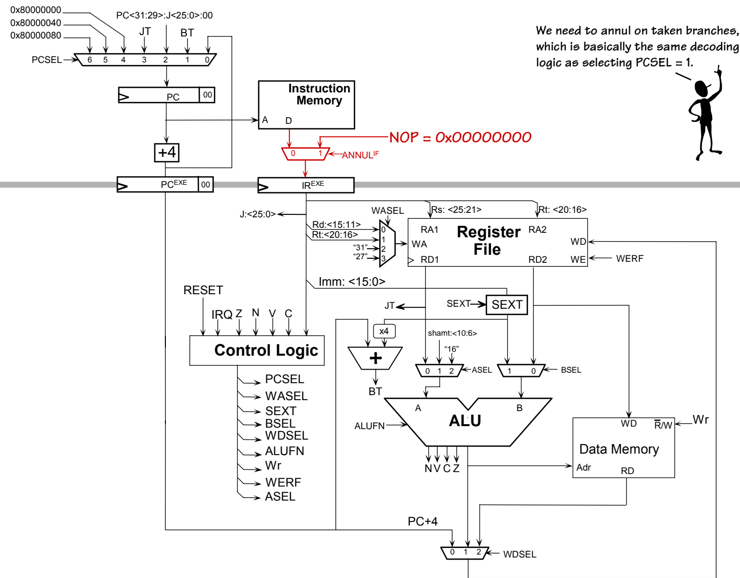
```
loop:  add  $t1,$t1,$t0
       srl  $t2,$t2,1
       bne  $t2,$0,loop
       andi $t0,$t2,1
       ...
```



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



We need to annul on taken branches, which is basically the same decoding logic as selecting PCSEL = 1.

Branch Alternative 2a

Always fill branch delay slots with NOP instructions (i.e., the software equivalent of alternative 1)

```
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	srl	bne	nop	add	cmp

Branch taken (at i+3)

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

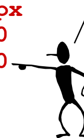
Cons: NOPs make code longer; 12% of cycles spent executing NOPs

Branch Alternative 2b(i)

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

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

This silly instruction UNDOES the effect of that last ADD



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

Branch taken (at i+3)

Pros: only two "extra" instructions are executed (on last iteration)

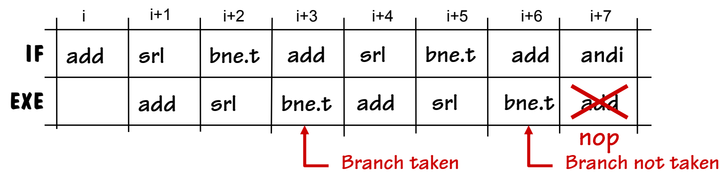
Cons: finding "useful" instructions that should *always* be executed is difficult; clever rewrite may be required. Program executes differently on unpipelined implementation.

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
      ...
```



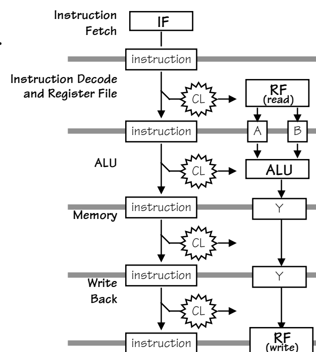
	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7
IF	add	srl	bne.t	add	srl	bne.t	add	andi
EXE		add	srl	bne.t	add	srl	bne.t	~~add~~ nop

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



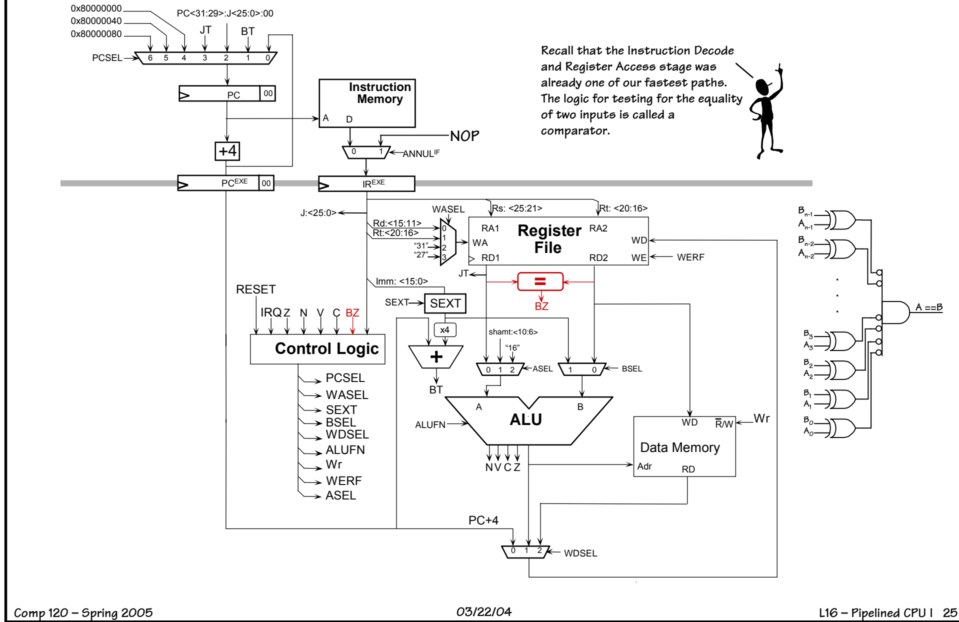
Where is the branch decision resolved?

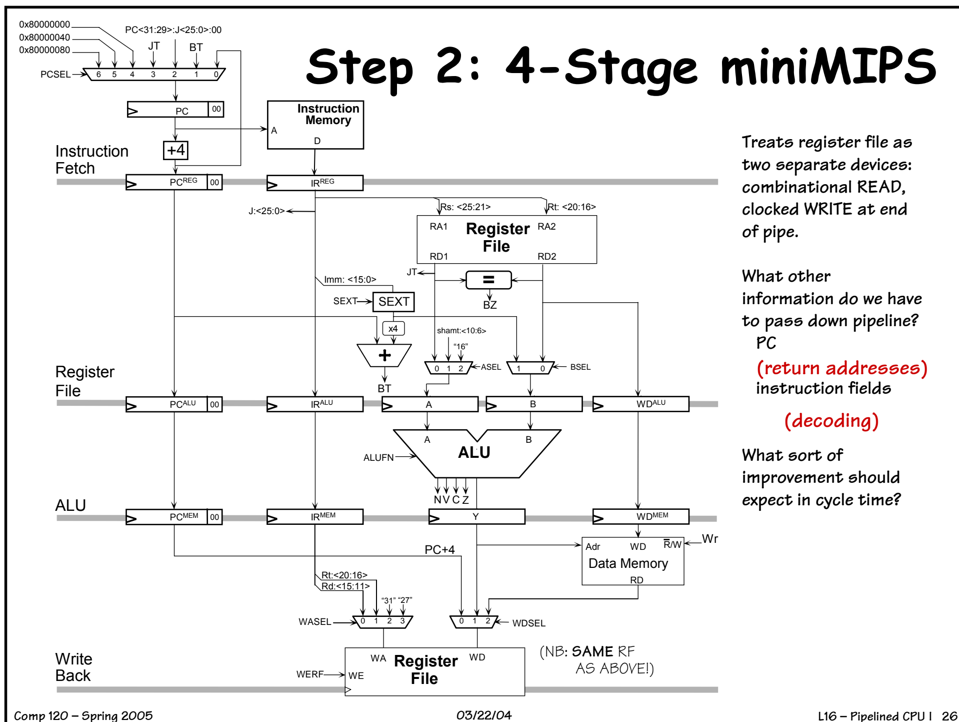
```
beq rs,rt,offset
```

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 miniMIPS implementation has at least TWO branch delay slots.

Is there any way we could make miniMIPS' 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
Recall that the Instruction Decode and Register Access stage was already one of our fastest paths. The logic for testing for the equality of two inputs is called a comparator.
Instruction Memory
Register File
Control Logic
ALU
Data Memory
NOP
RESET
PC+4
WDSEL
Comp 120 – Spring 2005
03/22/04
L16 – Pipelined CPU I 25


Step 2: 4-Stage miniMIPS
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?
Instruction Fetch
Register File
ALU
Write Back
(NB: SAME RF AS ABOVE!)
Instruction Memory
Data Memory
Comp 120 – Spring 2005
03/22/04
L16 – Pipelined CPU I 26

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:

TIME (cycles) →

Pipeline ↓

	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 "Structural Hazard"

BUT consider instead:

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

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

Before, we forbade feedback. Can't do that with a useful CPU.
How can we fix this one?

	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] 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:

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 mean "wasted" cycles. Lot's of wasted cycles. (A large percentage of instructions depend on results from the immediately preceding instruction)

Data Hazard Solution 3

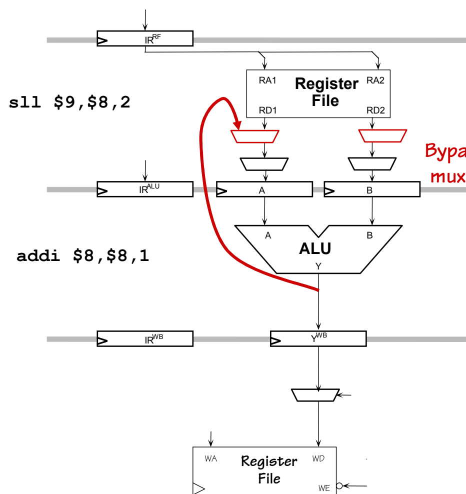
Bypass (aka forwarding) Paths:

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 used by `sll` in the RF stage!

Bypass Paths (I)



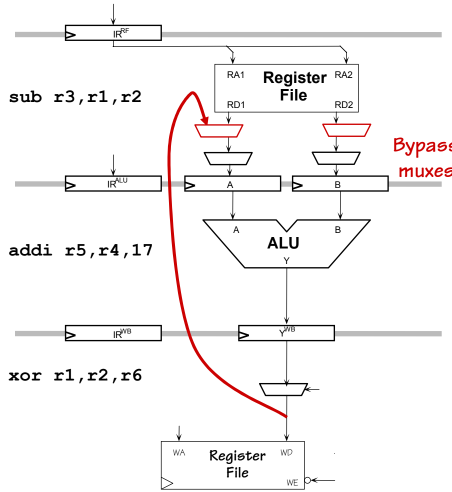
SELECT *this* BYPASS path if

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

i.e., instructions which use ALU to compute result

<u>and</u> Rs^{RF} != 0

Bypass Paths (II)



SELECT this BYPASS path if

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

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

Next Time