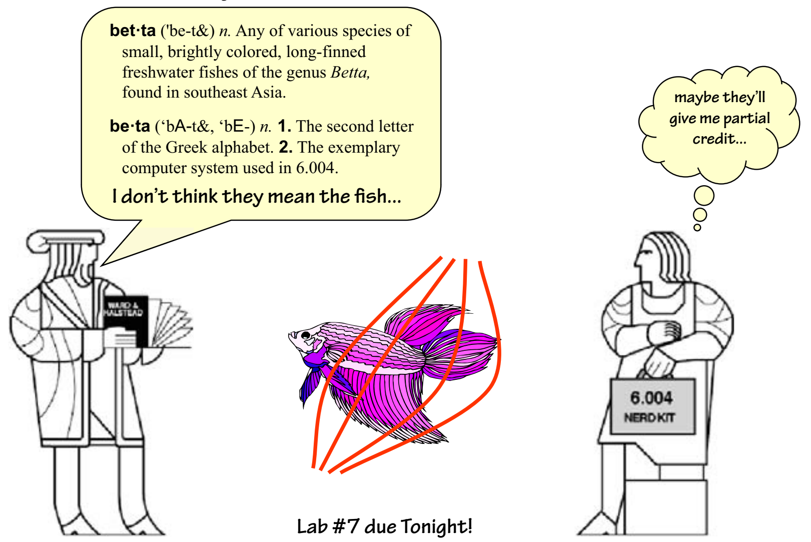
Pipelining the Beta


bet·ta ('be-t&) *n.* Any of various species of small, brightly colored, long-finned freshwater fishes of the genus *Betta,* found in southeast Asia.
be·ta ('bA-t&, 'bE-) *n.* **1.** The second letter of the Greek alphabet. **2.** The exemplary computer system used in 6.004.
I don't think they mean the fish...
maybe they'll give me partial credit...
WARD & HALSTEAD
6.004 NERD KIT

Lab #7 due Tonight!

CPU Performance

We've got a working Beta… can we make it *fast?*

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

MIPS = Millions of Instructions/Second

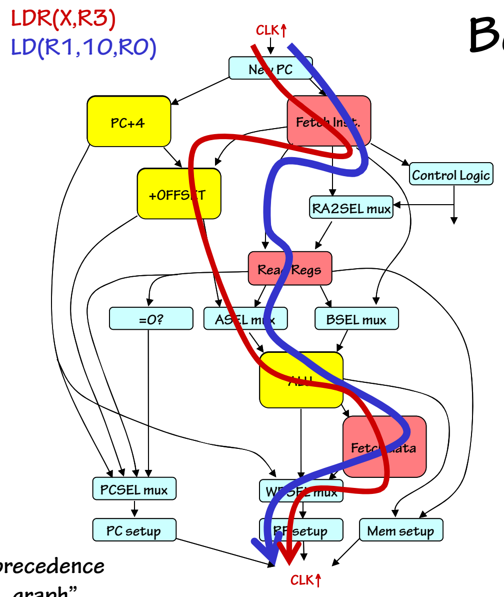
Freq = Clock Frequency, MHz

CPI = Clocks per Instruction

To Increase MIPS:

1. DECREASE CPI.
   - RISC *simplicity* reduces CPI to 1.0.
   - CPI *below* 1.0? Tough... you'll see multiple instruction issue machines in 6.823.
2. INCREASE Freq.
   - Freq limited by delay along longest combinational path; hence
   - *PIPELINING* is the key to improved performance through fast clocks.

Beta Timing



"precedence graph"

Wanted:
  longest paths

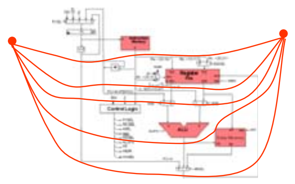
Complications:

- some apparent paths aren't "possible"
- operations have variable execution times (eg, ALU)
- time axis is not to scale (eg, $t_{PD,MEM}$ is very big!)

Why isn't this a 20-minute lecture?

We've learned how to pipeline combinational circuits.

What's the big deal?



1. The Beta isn't combinational…
   - *Explicit state* in register file, memory;
   - *Hidden* state in PC.
2. Consecutive operations – instruction executions – interact:
   - Jumps, branches dynamically change instruction sequence
   - Communication through registers, memory

Our goals:

- Move slow components into separate pipeline stages, running clock faster
- Maintain instruction semantics of unpipelined Beta as far as possible

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:

$$\begin{tikzcd}
\boxed{\text{IF}} \arrow[d] \\
\boxed{\text{RF}} \arrow[d] \\
\boxed{\text{ALU}} \arrow[d] \\
\boxed{\text{MEM}} \arrow[d] \\
\boxed{\text{WB}}
\end{tikzcd}$$

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

Register File stage: Reads source operands from register file, passes them to

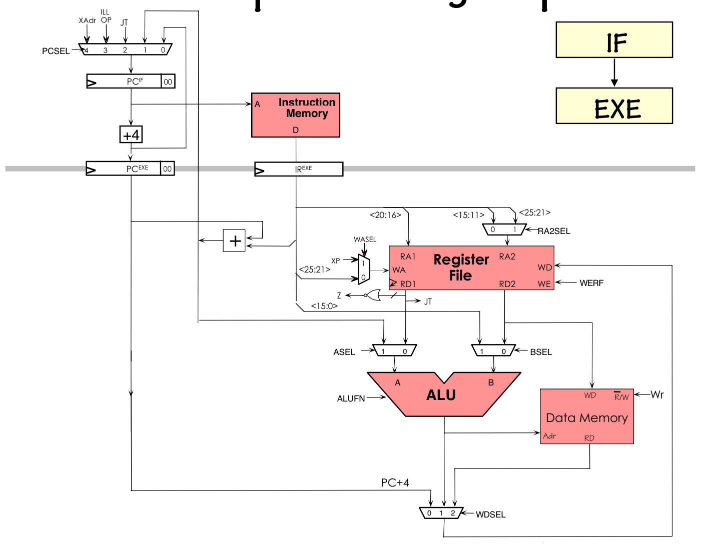
ALU stage: Performs indicated operation, passes result to

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

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

First Steps:

A Simple 2-Stage Pipeline



$$\begin{tikzcd}
\boxed{\text{IF}} \arrow[d] \\
\boxed{\text{EXE}}
\end{tikzcd}$$

2-Stage Pipelined Beta Operation

Consider a sequence of instructions:

```
. .
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
. . .
```

Executed on our 2-stage pipeline:

TIME (cycles) →

Pipeline ↓

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADDC	SUBC	XOR	MUL	...		
EXE		ADDC	SUBC	XOR	MUL	...	

Pipeline Control Hazards

BUT consider instead:

```
LOOP: ADD(r1, r3, r3)
      CMPLEC(r3, 100, r0)
      BT(r0, LOOP)
      XOR(r3, -1, r3)
      MUL(r1, r2, r2)
      . . .
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	XOR	...		
EXE		ADD	CMP	BT	?	...	

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!

Branch Delay Slots

PROBLEM: One (or more) following instructions have been pre-fetched by the time a branch is taken.

POSSIBLE SOLUTIONS:

1. Make hardware "annul" instructions following branches which are taken, e.g., by disabling WERF and WR.
2. "Program around it". Either
   a) Follow each BR 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

NOP = 'no-operation', e.g. ADD(R31, R31, R31)

Branch Alternative 1

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

```
LOOP:  ADD(r1, r3, r3)
       CMPLEC(r3, 100, r0)
       BT(r0, LOOP)
       XOR(r3, -1, r3)
       MUL(r1, r2, r2)
       ...
```

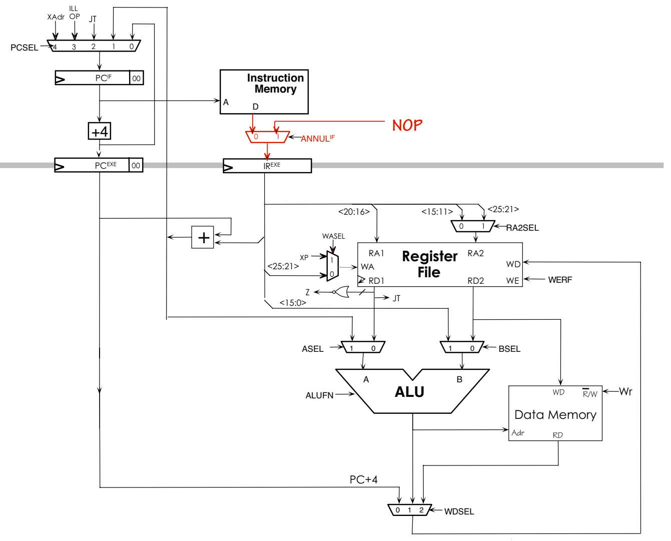
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	XOR	ADD	CMP	BT
EXE		ADD	CMP	BT	~~XOR~~ NOP	ADD	CMP

Branch taken

Pros: same program runs on both unpipelined and pipelined hardware

Cons: in SPEC benchmarks 14% of instructions are taken branches → 12% of total cycles are annulled

Branch Annulment Hardware



Branch Alternative 2a

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

```
LOOP:  ADD(r1, r3, r3)
       CMPLEC(r3, 100, r0)
       BT(r0, LOOP)
       NOP()
       XOR(r3, -1, r3)
       MUL(r1, r2, r2)
       ...
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	NOP	ADD	CMP	BT
EXE		ADD	CMP	BT	NOP	ADD	CMP

Branch taken

Pros: same as alternative 1

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

Branch Alternative 2b(i)

We need to add this silly instruction to UNDO the effects of that last ADD

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

```
LOOP:  ADD(r1,r3,r3)
LOOPx: CMPLEC(r3,100,r0)
       BT(r0,LOOPx)
       ADD(r1,r3,r3)
       SUB(r3,r1,r3)
       XOR(r3,-1,r3)
       MUL(r1,r2,r2)
       ...
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	ADD	CMP	BT	ADD
EXE		ADD	CMP	BT	ADD	CMP	BT

Branch taken

Pros: only two "extra" instructions are executed (on last iteration)
Cons: finding "useful" instructions that are *always* executed is difficult; clever rewrite may be required. Program executes differently on naïve unpipelined implementation.

Branch Alternative 2b(ii)

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

```
LOOP:  ADD(r1, r3, r3)
LOOPx: CMPLEC(r3, 100, r0)
       BT.taken(r0, LOOPx)
       ADD(r1, r3, r3)
       XOR(r3, -1, r3)
       MUL(r1, r2, r2)
       ...
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	BT	ADD	CMP	BT	ADD
EXE		ADD	CMP	BT	ADD	CMP	BT

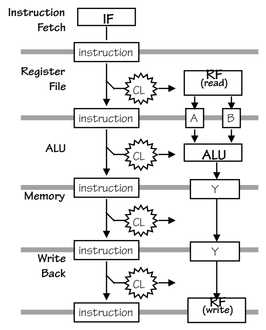
Branch taken

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; not really useful with more than one delay slot.

Architectural Issue: Branch Decision Timing

BETA approach:
- SIMPLE branch condition logic ... Test for Reg[Ra] = 0!
- ADVANTAGE: early decision, single delay slot

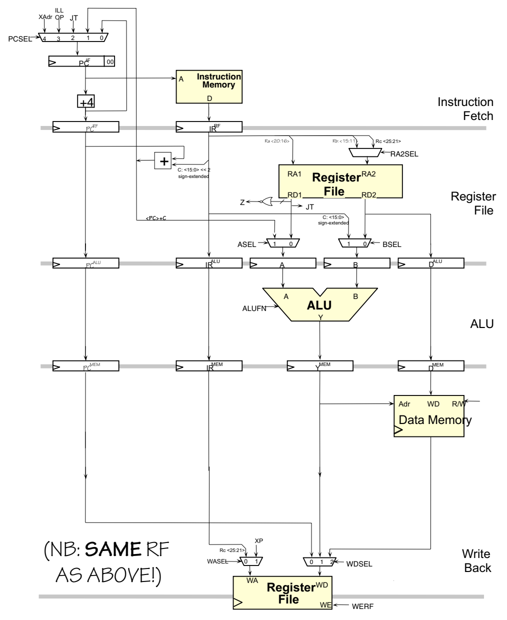
ALTERNATIVES:
- Compare-and-branch... (eg, if Reg[Ra] > Reg[Rb])
- MORE powerful, but
- LATER decision (hence more delay slots)



Wow! I guess those guys really were thinking when they made up all those instructions

Suppose decision were made in the ALU stage ... then there would be 2 branch delay slots (and instructions to annul!)

4-Stage Beta Pipeline



(NB: SAME RF AS ABOVE!)

Treat 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 Beta Operation

Consider a sequence of instructions:

```
...
ADDC(r1, 1, r2)
SUBC(r1, 1, r3)
XOR(r1, r5, r1)
MUL(r2, r6, r0)
...
```

Executed on our 4-stage pipeline:

TIME (cycles) →

Pipeline ↓

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADDC	SUBC	XOR	MUL	...		
RF		ADDC	SUBC	XOR	MUL	...	
ALU			ADDC	SUBC	XOR	MUL	...
WB				ADDC	SUBC	XOR	MUL

Pipeline "Data Hazard"

BUT consider instead:

```
ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)
MULC(r1, 100, r4)
SUB(r1, r2, r5)
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	CMP	MUL	SUB			
RF		ADD	CMP	MUL	SUB		
ALU			ADD	CMP	MUL	SUB	
WB				ADD	CMP	MUL	SUB

r3 fetched

r3 available

Oops! CMP is trying to read Reg[R3] during cycle i+2 but ADD doesn't write its result into Reg[R3] 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!
- Costs code efficiency.

EXAMPLE: Rewrite

```
ADD(r1, r2, r3)                    ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)                MULC(r1, 100, r4)
MULC(r1, 100, r4)        as        SUB(r1, r2, r5)
SUB(r1, r2, r5)                    CMPLEC(r3, 100, r0)
```

How often can we do this?

Programmer's fallback: Insert NOPs (sigh!)

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	ADD	CMP	MUL	*MUL*	*MUL*	SUB	
RF		ADD	CMP	*CMP*	*CMP*	MUL	SUB
ALU			ADD	NOP	NOP	CMP	MUL
WB				ADD	NOP	NOP	CMP

Drawback: NOPs mean "wasted" cycles

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	ADD	CMP	MUL	SUB			
RF		ADD	CMP	MUL	SUB		
ALU			ADD	CMP	MUL	SUB	
WB				ADD	CMP	MUL	SUB

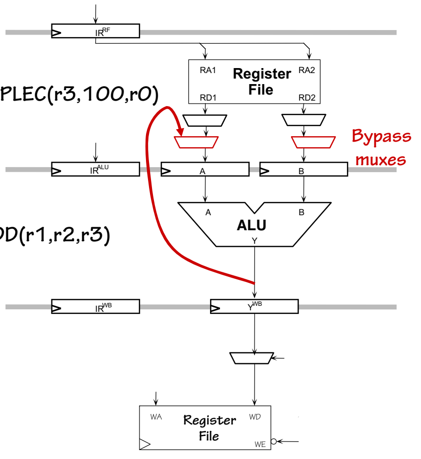
r3 available

Idea: the result from the ADD which will be written into the register file at the end of cycle I+3 is actually available at output of ALU during cycle I+2 – just in time for it to be used by CMP in the RF stage!

Bypass Paths (I)

CMPLEC(r3,100,r0)

ADD(r1,r2,r3)



SELECT *this* BYPASS path if
OpCode^{RF} = reads Ra
<u>and</u> OpCode^{ALU} = OP, OPC
<u>and</u> Ra^{RF} = Rc^{ALU}

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

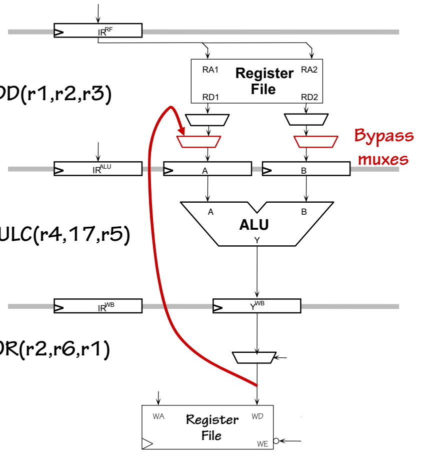
<u>and</u> Ra^{RF} != R31

Bypass Paths (II)

ADD(r1,r2,r3)

MULC(r4,17,r5)

XOR(r2,r6,r1)



SELECT *this* BYPASS path if
OpCode^{RF} = reads Ra
<u>and</u> Ra^{RF} != R31
<u>and</u> not using ALU bypass
<u>and</u> WERF = 1
<u>and</u> Ra^{RF} = WA

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

Next Time