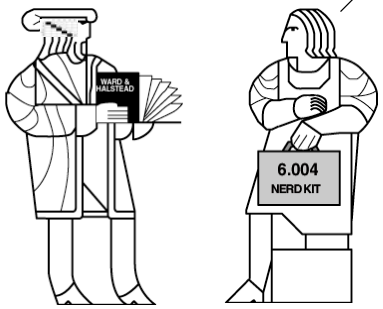
6.004 Computation Structures
Spring 2009

Pipeline Issues

This pipeline stuff makes my head hurt!

Maybe it's that dumb hat



Home Stretch:
Lab #8 due Thursday 5/7;
Quiz 5 FRIDAY 5/8!

Recalling Data Hazards

PROBLEM: Subsequent instructions can reference the contents of a register well before the pipeline stage where the register is written.

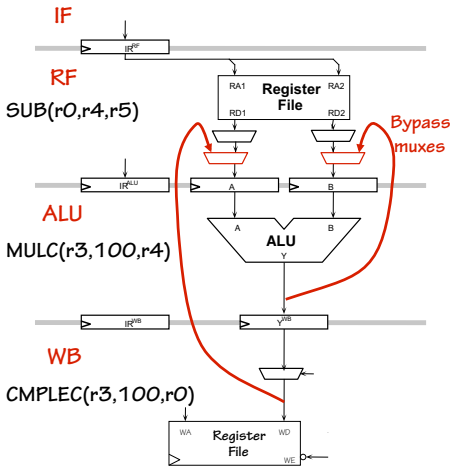
```
ADD(r1, r2, r3)
CMPLEC(r3, 100, r0)
MULC(r3, 100, r4)
SUB(r0, r4, r5)
```

	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

SOLUTION #1: Deal with it in SOFTWARE; expose the pipeline for all to see.

SOLUTION #2: Add special hardware to maintain the sequential execution semantics of the ISA.

Bypass Paths



Add special data paths, called BYPASSES, that route the results of the ALU and WB stages to the RF stage, thus substituting the register's old contents with a value that will be written to that register at some point in the future.

Detection of these cases has to be incorporated into the decoding logic of the RF stage, which basically looks at the instructions in the ALU and WB stage to see if their destination register matches a source register reference.

But there are some problems that BYPASSING CAN'T FIX!

Load Hazards

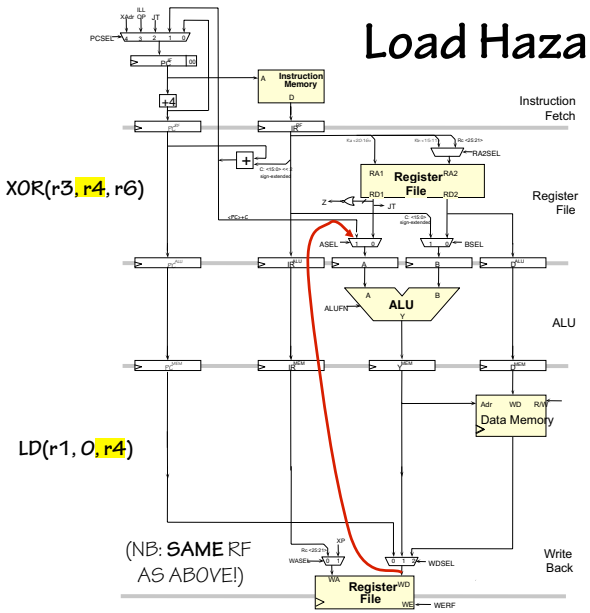
Consider LOADS:
Can we fix all these problems using bypass paths?

```
LD(r1, 0, r4)
ADD(r4, r1, r5)
XOR(r3, r4, r6)
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	LD	ADD	XOR				
RF		LD	ADD	XOR			
ALU			LD	ADD	XOR		
WB				LD	ADD	XOR	

The hazard between the XOR and the LD can be addressed by our established bypass paths…

Load Hazard (easy)



The XOR operand r4 can simply be bypassed from the output of the memory in the WB stage to the RF stage… by our normal bypass path.

Structural Data Hazard

The XOR hazard is pretty easy, but…

```
LD(r1, 0, r4)
ADD(r1, r4, r5)
XOR(r3, r4, r6)
```

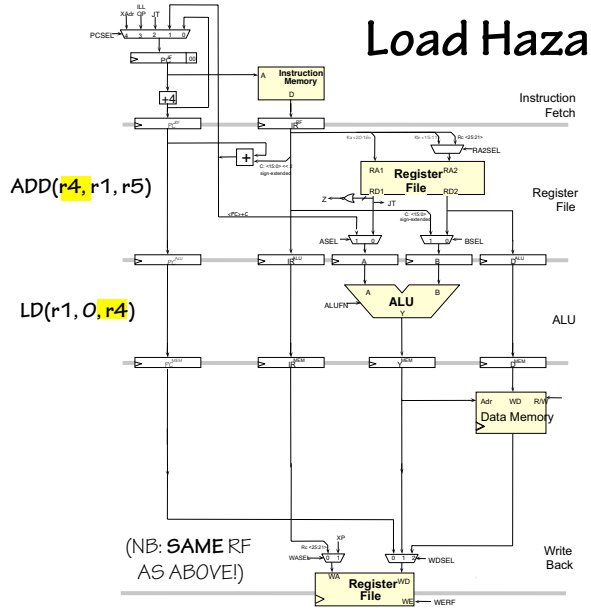
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	LD	ADD	XOR				
RF		LD	ADD	XOR			
ALU			LD	ADD	XOR		
WB				LD	ADD	XOR	

???

How do we fix this one?

In a 4-stage pipeline, for a LD instruction fetched during clock i, the data from memory isn't returned from memory until late into cycle i+3. Bypassing can fix the XOR but not ADD!

Load Hazard (hard)



The r4 operand to the ADD instruction hasn't yet been fetched from memory.

It exists NOWHERE on our data paths – we can't solve this problem by bypassing!

Load Delay

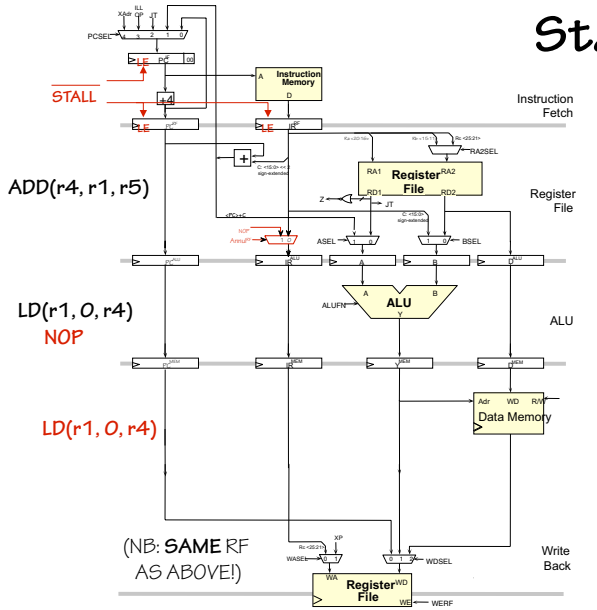
Bypassing can't fix the problem with ADD since the data simply isn't available! We have to add some *pipeline interlock hardware* to stall ADD's execution.

```
LD(r1, 0, r4)
ADD(r1, r4, r5)
XOR(r3, r4, r6)
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	LD	ADD	XOR	*XOR*			
RF		LD	ADD	*ADD*	XOR		
ALU			LD	NOP	ADD	XOR	
WB				LD	NOP	ADD	XOR

If the compiler knows about a machine's load delay, it can often rearrange code sequences to eliminate such hazards. Many compilers provide machine-specific *instruction scheduling*.

Stall Logic



(1) Freeze IF, RF stages
(2) Introduce NOP into ALU stage
(3) Wait until operand is available

Memory Timing & Pipelining

But, but, what about FASTER processors?

FACT: Processors have become very fast relative to memories! And this gap continues to grow…

Do we just increase the clock period to accommodate this bottleneck component?

ALTERNATIVE: Longer pipelines.

1. Add "MEMORY WAIT" stages between START of read operation & return of data.
2. Build pipelined memories, so that multiple (say, N) memory transactions can be in progress at once.

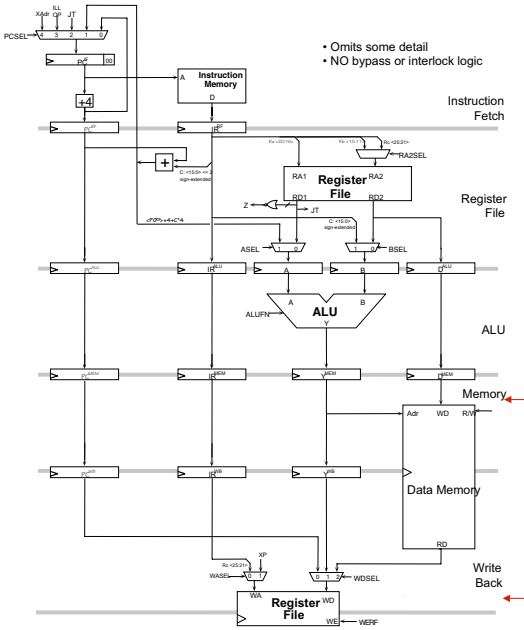
These steps add load delay slots; hence

3. Stall pipeline on unbypassable load delays.

A 4-Stage pipeline requires READ access in less than one clock.

A 5-Stage pipeline would allow nearly two clocks...

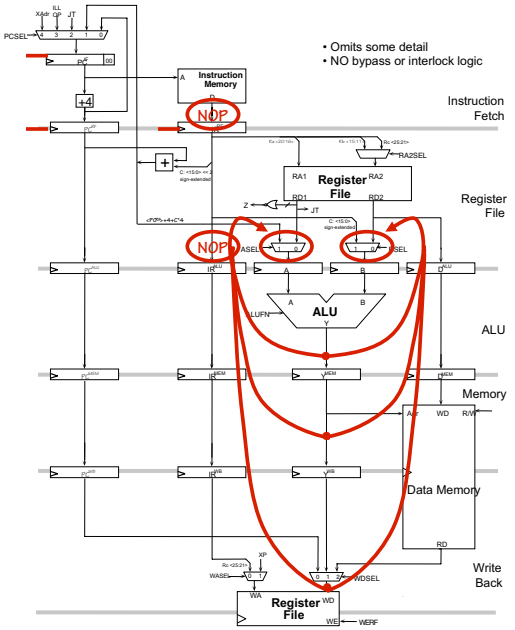
5-stage Pipeline



Address available right after instruction enters Memory pipe stage

almost 2 clock cycles

Data needed right before rising clock edge at end of Write Back pipe stage



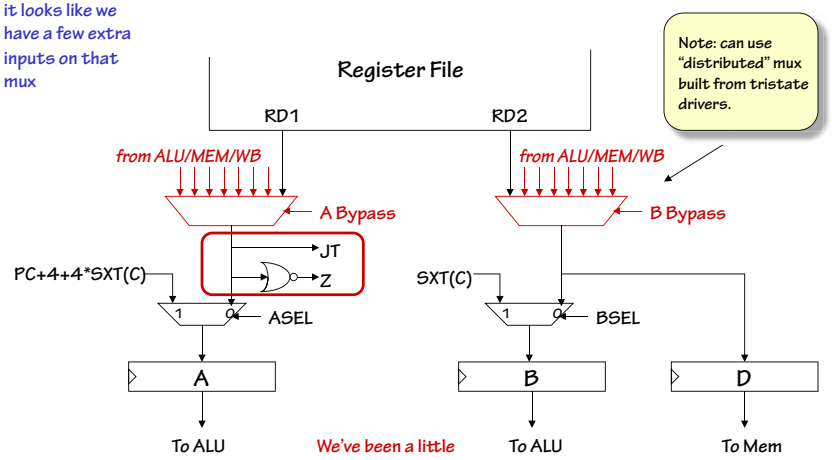
5-stage pipeline

We wanted a simple, clean pipeline but…

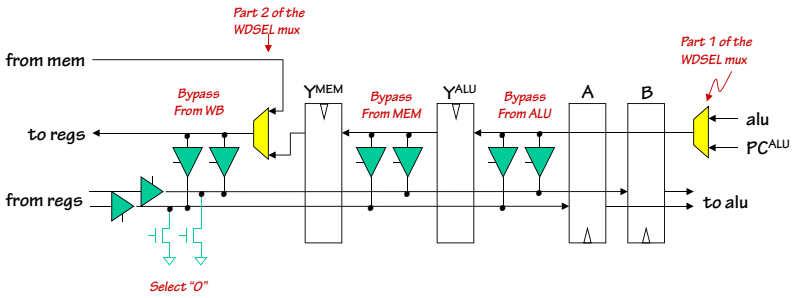
- added IR^{IF} mux to annul branch-slot instructions
- added A/B bypass muxes to get data before it's written to regfile
- added LE/muxes to freeze IF/RF stage so we can wait for LD to reach WB stage

RF-stage Bypass Details

1, 2, 3, 4… Hum, it looks like we have a few extra inputs on that mux



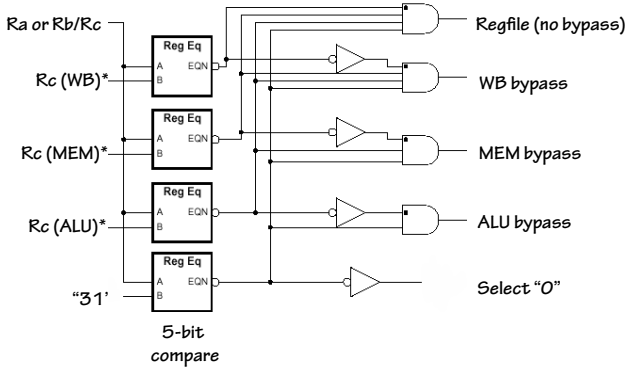
Bypass Implementation



To reduce the amount of bypass logic, the WDSEL mux has been split: choice between ALU and PC+4 is made in ALU stage, choice between ALU/PC and MEMDATA is made is WB stage.

Bypass Logic

Beta Bypass logic (need two copies for A/B data):



* If instruction is a ST (doesn't write into regfile), set RC for ALU/MEM/WB to R31

Linkage Register Write Timing

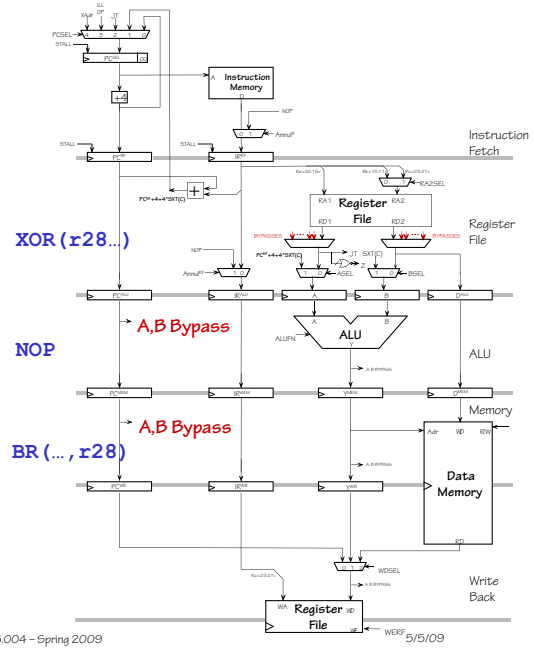
```
| The code: Assume Reg[LP] = 100...
        ADD(r31, r31, LP)
        BR(f, LP)          | Reg[LP] = .+4
x:      SUB(LP, 1, LP)
        ...
f:      XOR(LP, r31, r0)
        OR(r31, LP, r1)
        ADD(r31, LP, r2)
```

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	BR	SUB	XOR	OR	ADD	
RF		ADD	BR	NOP	XOR	OR	ADD
ALU			ADD	BR	NOP	XOR	OR
MEM				ADD	BR	NOP	XOR
WB					ADD	BR	NOP

BR Decision Time — ADD writes — BR writes

Can we make XOR's regfile access work by bypassing?

BR/JMP PC bypass



For BR/JMP, Rc value is taken from PC, not ALU.

So we have to add bypass paths for PC^{ALU} and PC^{MEM}.

PC^{WB} is already taken care of if we bypass WB stage from output of WDSEL mux.

Unused Opcode Traps

IDEA: TRAP illegal instructions to a special routine in the Operating System, which can

- Interpret them in software; or
- Print humane error report.

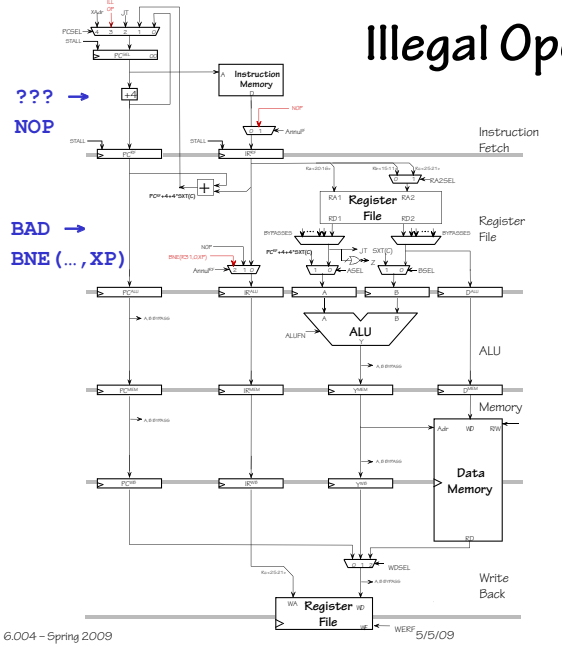
IMPLEMENTATION: On Bad Opcode (discovered in RF Stage):

- Select IllOp adr as next PC
- Annul instruction in IF stage
- Substitute BNE(r31,0,XP) for bad instruction - will (eventually) store PC+4 into XP ... need bypass paths to make XP usable immediately by code at IllOp!

```
| User program:
        ...
        BAD(...)      | Illegal instr.
r:      ...

| Operating System: UUO Handler
IllOp:  ST(r0,...)    | Save a reg,
        LD(xp,-4,r0)  | Fetch bad instr
        ...
        LD(...,r0)    | Restore regs,
        JMP(xp)       | Return to pgm.
```


Illegal Opcode Traps



Bad opcode decoded in RF stage:

- PC ← address of IllOp handler
- Annul instruction in IF
- Force BNE(R31,0,XP) in RF stage → will save PC+4 in XP when it reaches WB stage

Taking Exception...

In general, we'd like to annul ALL instructions following one that causes a trap or fault:

FREEZE state at time of exception, for inspection by handler code.

ILLEGAL INSTRUCTIONS are recognizable in RF stage of pipe; are ALL faults & traps?

CONSIDER:

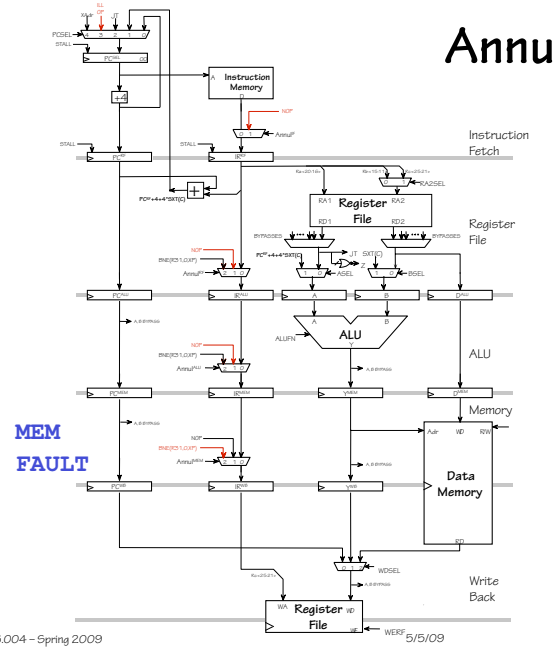
ARITHMETIC EXCEPTIONS: divide by zero, etc.

- Caught by ALU subsystem, during processing of data in ALU stage

MEMORY FAULTS: Program reference to illegal memory location...

- Caught by MEMORY subsystem, during processing of address input in MEM stage

Annulment Logic



Fault in ??? stage:

- PC ← address of fault handler
- Force BNE(R31,0,XP) in ??? stage → will save PC+4 in XP when it reaches WB stage
- Annul all *following* instructions (those earlier in the pipeline): called "flushing the pipe"

Asynchronous I/O Interrupts

This should be easy.

Take, for example,

```
| The interrupted code:
        ...
        ADD(...)      <-- Interrupt Taken HERE
        SUB(...)
        MUL(...)
        XOR(...)
        ...

| The interrupt handler:
xh:     OR(...)
        ...
        JMP(xp)
```

Suppose key struck, interrupt requested (via IRQ) during the fetch of ADD. Then let's

- Select XAdr (handler) as next PC
- Leave ADD in pipeline; NO annulment!
- Code handler to return to SUB instruction.

Can this work???

Let's find out…

Asynchronous Interrupt Timing

```
        ...
        ADD(...)      <-- Interrupt Taken HERE
        SUB(...)
        MUL(...)
        XOR(...)
        ...
xh:     OR(...)       | interrupt handler
        ...
        JMP(xp)
```

Interrupt Occurs PC = Xadr??

	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	OR	…	…			
RF		ADD	OR	…			
ALU			ADD	OR	…		
MEM				ADD	OR	…	
WB					ADD	OR	…

PROBLEM: When does old PC+4 get written to XP???

Making Interrupts Work

Alternative: When taking interrupt,

- ANNUL instruction in IF stage… BUT

  instead of changing it to a NOP, change it to BNE(r31,0,XP)

  This will cause PC+4 of annulled instruction to be written to XP!

- CODE HANDLER to return to Reg[XP]-4 (since the annulled instruction is never executed)

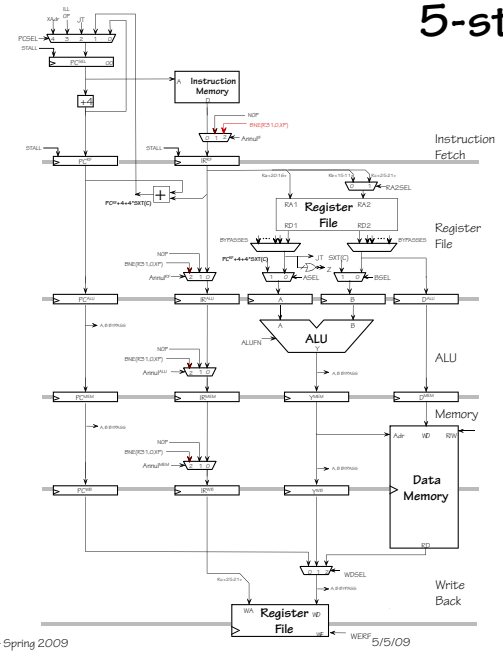
	i	i+1	i+2	i+3	i+4	i+5	i+6
IF	ADD	OR	…	…			
RF		BNE	OR	…			
ALU			BNE	OR	…		
MEM				BNE	OR	…	
WB					BNE	OR	

Interrupt taken

"Smart" Interrupt Handler

```
| The interrupted code:
        ...
        ADD(...)        <-- Interrupt taken HERE, ADD instruction annulled
        SUB(...)
        MUL(...)
        XOR(...)
        ...
| The interrupt handler:
xh:     OR(...)
        ...
        SUBC(xp,4,xp)          | Adjust XP so
        we
        JMP(xp)           | return to annulled
                          | instruction (ADD)
```


5-stage Pipeline: Final Version



- Can annul instruction at each stage
- Can force instruction to BNE(R31,0,XP) in each stage → will save PC+4 in XP when it reaches WB stage
- Can stall IF and RF stages while waiting for LD result to reach WB
- Can bypass results from ALU, MEM and WB back to RF

Pipeline Review

Simple unpipelined Beta:
- 1 cycle/instruction
- long cycle time: mem+regs+alu +mem

2-Stage pipeline:
- increased throughput (<2x)
- introduced branch delay slots
  - Choice of executing or annulling inst. after branch

5-stage pipeline:
- increased throughput (3x???)
- branch delay slots
- delayed register writeback (3 cycles)
  - Add bypass paths (10) to access correct value

- memory data available only in WB stage
  - Introduce NOPs at IR^{ALU}, stall IF and RF stages until LD result ready
- handle RF/ALU/MEM stage exceptions
  - Save PC+4 in XP (fake a BR)
  - annul following insts. (those earlier in pipeline)
- implement interrupts
  - Throw away IF inst., save PC+4 in XP, fix return
- extra HW due to pipelining
  - Registers to hold values between stages
  - Data bypass muxes in RF stage
  - Inst. muxes for "rewriting" code to annul or save PC

RISC = Simplicity???

"The P.T. Barnum World's Tallest Dwarf Competition"

World's Most Complex RISC?