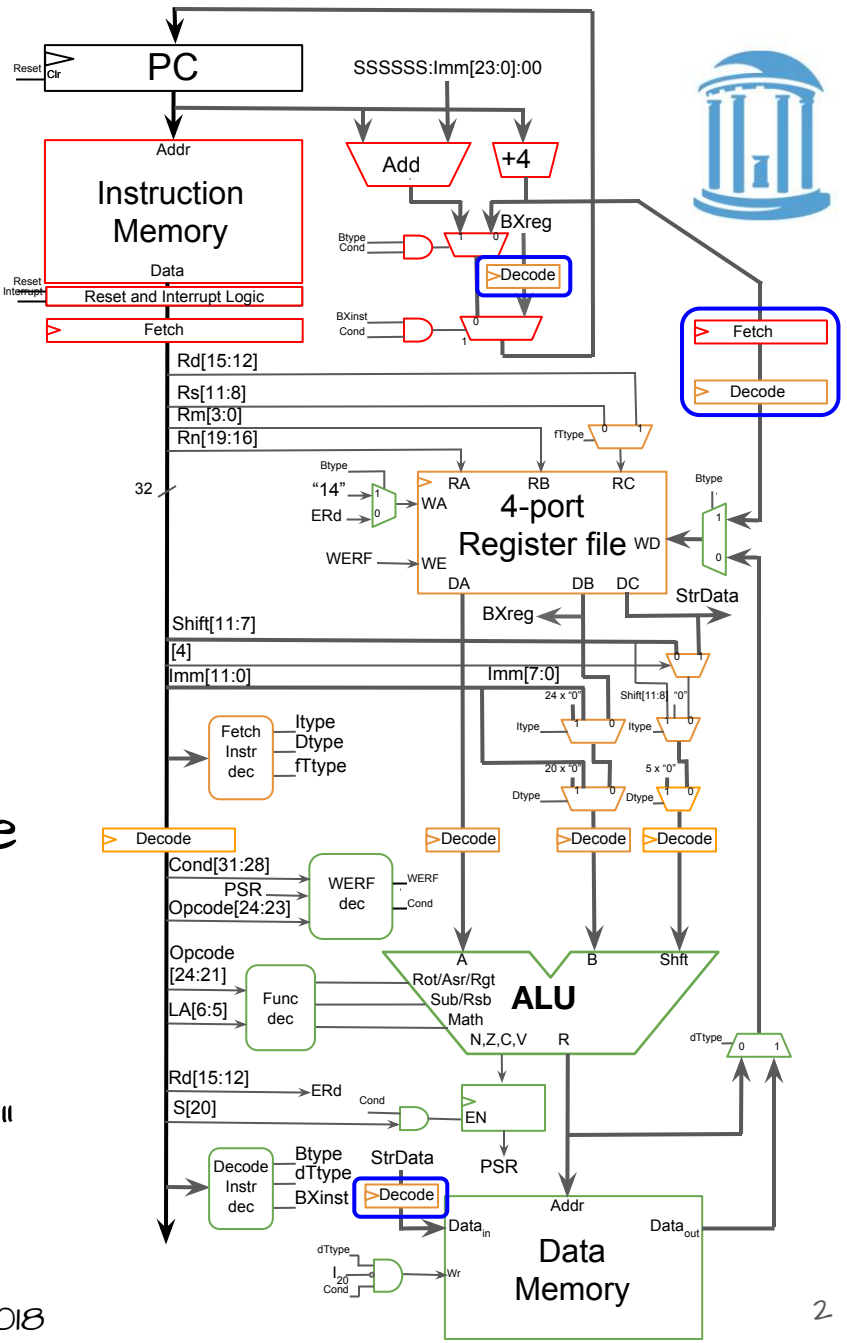
Pipeline Hazards

- Short lecture with
  8th and final lab on 11/30
- 5th and final problem due on 12/1

ARM 3-STAGE PIPELINE

- Fetch, Decode, and Execute Stages
- Instructions are decoded in both the Fetch and Decode stages
- Register ports are "Read" in the Decode stage and "Written" at in the end of the Execute stage
- PC+4, register reads for stores (StrData). and BX source (BXreg) are "delayed" for use in later stages



Simple Instruction Flow

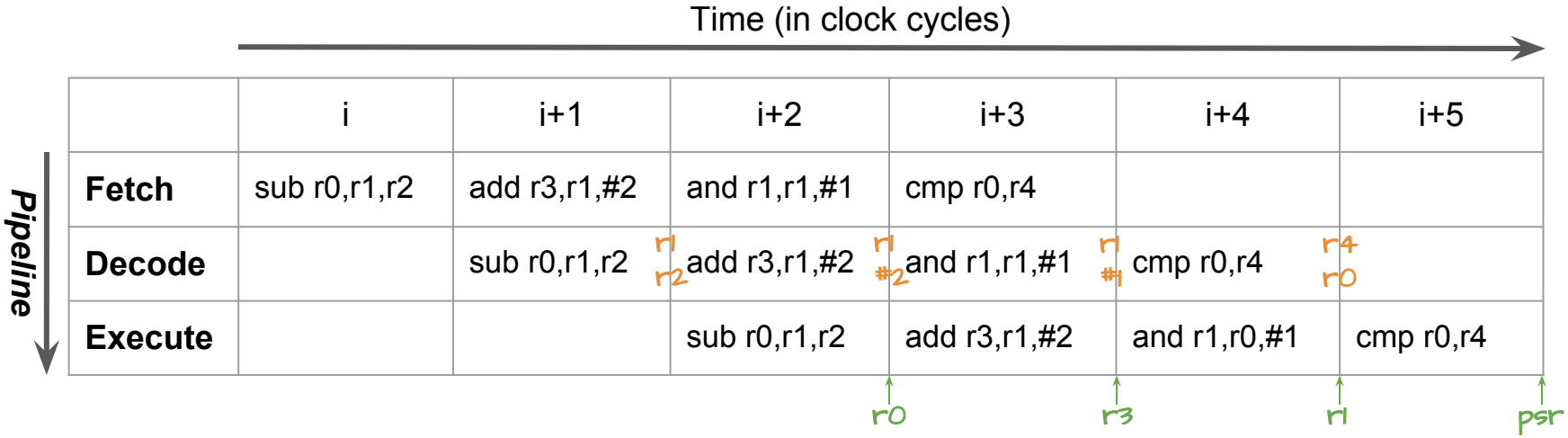
Consider the following instruction sequence:

Instruction becomes available at the end of the Fetch stage

Operands at the end of Decode

Destination and PSR are updated at the end of Execute

```
...
sub     r0,r1,r2
add     r3,r1,#2
and     r1,r1,#1
cmp     r0,r4
```



Pipeline Control Hazards

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

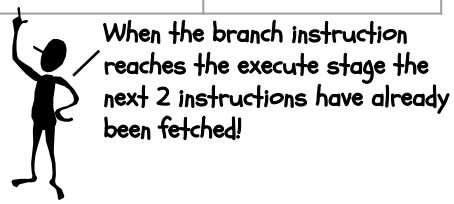
Consider the instruction sequence shown:

```
...
loop:   add   r0,r0,r0
        cmp   r0,#64
        ble   loop
        and   r1,r0,#7
        sub   r1,r0,r1
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	sub r1,r0,r1	add r0,r0,r0
Decode		add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	???
Execute			add r0,r0,r0	cmp r0,#64	ble loop	???



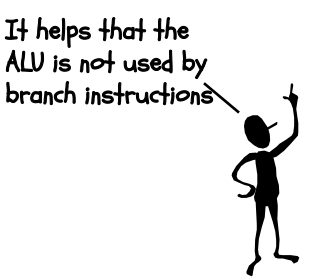
Branch Fixes

Problem: Two instructions following a branch are fetched before the branch decision is made (to take or not to take)

Solutions:

1. Program around it. Define the ISA such that the branch does not take effect until after instructions in the "DELAY SLOTS" complete. This is how MIPS pipelines work. It leads to ODD looking code in tight (short) loops. Of course you could always put NOPs in the delay slots.
2. Detect the branch decision as early as possible, and ANNUL instructions in the delay slots. This is what ARM does.

Early Detect and Annul



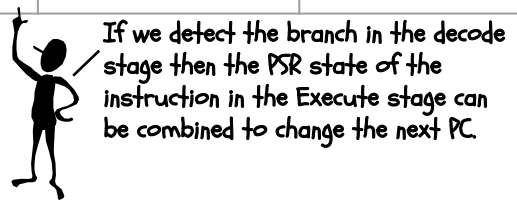
We can detect branch instructions (B, BL, or BX) in the Decode stage. The decision to branch is decided no later than the current instruction in the Execute stage. Thus, we could make The branch decision in the Decode stage. We then annul the following instruction by disabling WERF and PSR updates! Making the next instruction a NOP!

```
...
loop:   add   r0,r0,r0
        cmp   r0,#64
        ble   loop
        and   r1,r0,#7
        sub   r1,r0,r1
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	add r0,r0,r0	cmp r0,#64
Decode		add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	add r0,r0,r0
Execute			add r0,r0,r0	cmp r0,#64	ble loop	**NOP**



The cost of taken branches

When an ARM branch is ***taken*** the branch instructions are effectively **2 cycles rather than 1** when they aren't. In a MIPS-like instruction set, one can often fill the delay slots with useful instructions, but they are executed whether or not the branch is taken.

The ARM approach is easier to understand, and since it does not "EXPOSE" the pipeline, it also allows for an alternative number of pipeline stages to be implemented in future designs, while conserving code compatibility.

Lastly, using ARM, many **conditional branches can be eliminated using the condition execution**, which pipelines beautifully!

Structural Pipeline Hazards

There's another problem with our code fragment!

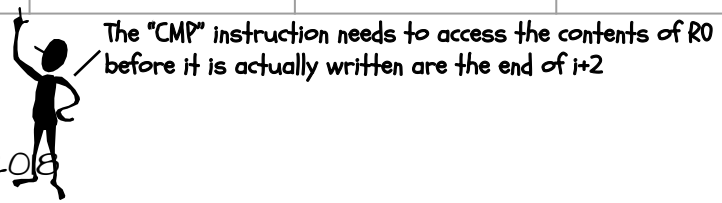
The destination register of instructions are written at the end of the Execute stage. However the following instruction might use this result as a source operand.

```
...
loop:  add   r0,r0,r0
       cmp   r0,#64
       ble   loop
       and   r1,r0,#7
       sub   r1,r0,r1
```

Time (in clock cycles)

Pipeline

	i	i+1	i+2	i+3	i+4	i+5
Fetch	add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	add r0,r0,r0	cmp r0,#64
Decode		add r0,r0,r0	cmp r0,#64	ble loop	and r1,r0,#7	add r0,r0,r0
Execute			add r0,r0,r0	cmp r0,#64	ble loop	NOP



Data Hazards

Problem: When a register source is needed from a later stage of the pipeline before it is written.

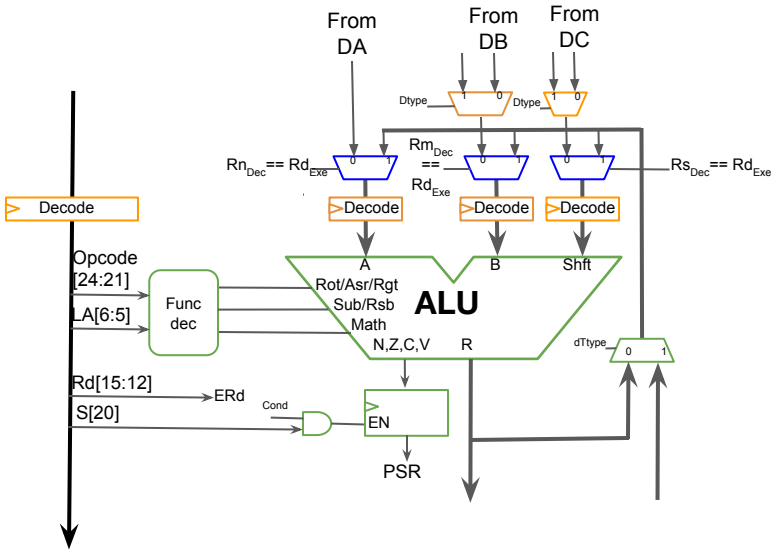
Solutions:

1. Program around it. One could document the weird semantics-- "You can't reference the destination register of an instruction in the immediately following instruction." Would make make assembly language even harder to understand. Would expose the pipeline, once again making future improvements difficult to implement while maintaining code compatibility.
2. Hardware bypass multiplexers.

Source Bypassing

The idea here is to load the value that to be saved in the destination register also into the pipeline registers that hold the ALU operands.

We also need bypass MUXes on the StrReg and BXreg pipeline registers.



Pipeline

	i	i+1	i+2
Fetch	add r0,r0,r0	cmp r0,#64	ble loop
Decode		add r0,r0,r0	cmp r0,#64
Execute			add r0,r0,r0

The new value for R0 will be computed just prior to the rising clock edge between i+2 and i+3, we can take the output of

Load/Store Stalls

Load and Store memory accesses are the actual **bottleneck** of the ARM pipeline. Also, recall that instructions and load/stores actually come from the same memory. Thus, we need to stall instruction fetching to allow for loads and stores.

```
...
loop:  ldr  r0,[r1,#4]
       add  r0,r0,#4
       str  r0,[r1,#4]
       sub  r0,r0,#4
       and  r2,r2,#f
```

Time (in clock cycles)

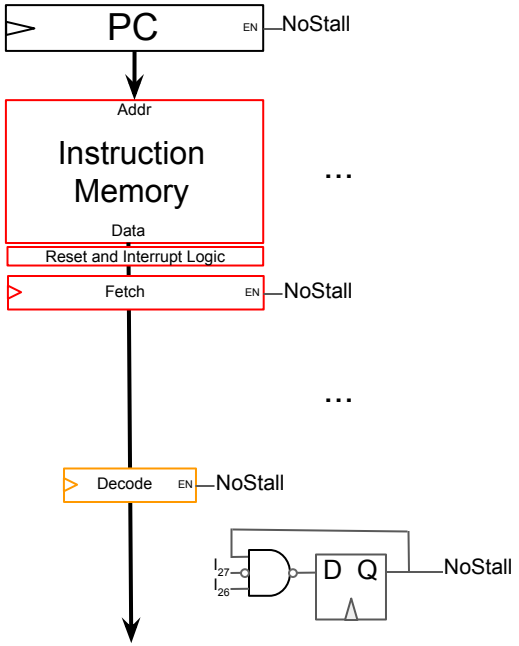
Pipeline

	i	i+1	i+2	i+3	i+4	i+5	i+6
Fetch	ldr r0,[r1,#4]	add r0,r0,#4	str r0,[r1,#4]	Mem read	sub r0,r0,#4	and r2,r2,#f	Mem store
Decode		ldr r0,[r1,#4]	add r0,r0,#4		str r0,[r1,#4]	sub r0,r0,#4	
Execute			ldr r0,[r1,#4]		add r0,r0,#4	str r0,[r1,#4]	

Load/Store stall implementation

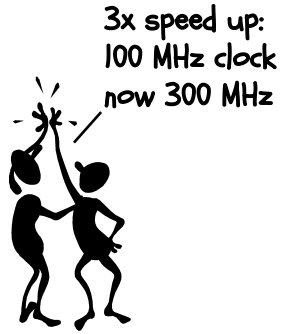
Disable loading of pipeline registers for one clock when a load or store instruction reaches the execute stage.

1. Adding enable lines to the PC and pipeline registers on the control path
2. A simple 2-state state machine to stall the pipeline for 1 state to allow for the load/store memory cycle.



Where does this leave us

Overall we can now nearly triple the clock rate. Instructions have a throughput of one-per-clock with the following caveats:



1. Taken branches take 2 cycles.
2. Loads and store take 2 cycles.

You can pipeline an ARM CPU even more. There exist ARM implementations with 7, 8, and 9 pipeline stages. But the overhead of bypass paths and stall cases increase.

Reality vs Specmanship

Assuming approximately 10% of instructions executed are branches, and of those 80% of the time they are taken, and 15% of instruction executed are loads or stores, what sort of real speed up do we expect?

$$Perf_{before} = (100) * 1 = 100 \text{ Clocks} * 10 * 10^{-9} \text{ sec/clock} = 1000 * 10^{-9} \text{ secs}$$

$$Perf_{after} = (10)((0.8) * 2 + (0.2) * 1) + 15 * 2 + 75 * 1 = 123 \text{ Clocks}$$

$$123 * 3.333 * 10^{-9} \text{ sec/clock} = 410 * 10^{-9} \text{ secs}$$

$$\text{Speedup} = \frac{Perf_{before}}{Perf_{after}} = 1000/410 = 2.439 \text{ X}$$

Next time

It appears memory access time is our real bottleneck. What tricks can be applied to improving CPU performance in this case?

- Interleaving
- Block-transfers
- Caching