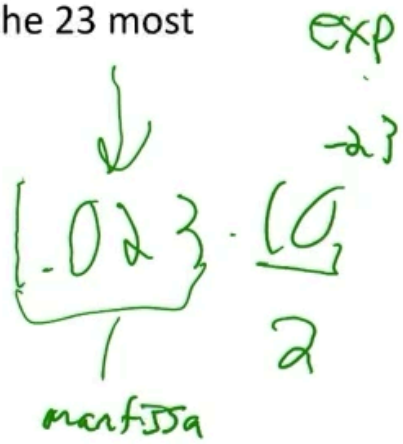
Review: old exams, homeworks, IA notes by daniel, review session (on eecs 370 website!)
Tips: don't spend too much time on one problem
Binary, octal, and hexadecimal conversions
2's complement representation

- To represent negative number: flip bits and add 1
- To represent positive number: subtract 1 and flip bits

Floating point formats

IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
  - -127 → 00000000 1 → 10000000
  - -126 → 00000001 2 → 10000001
  - … …
  - 0 → 01111111 128 → 11111111



- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes), Significand 0 (all zeroes), Sign + or -

- First bit is 0 if positive, 1 if negative
- Encode (exponent + 127) for the exponent
- Remember: everything is in base 2
- There is an implicit 1 before the mantissa

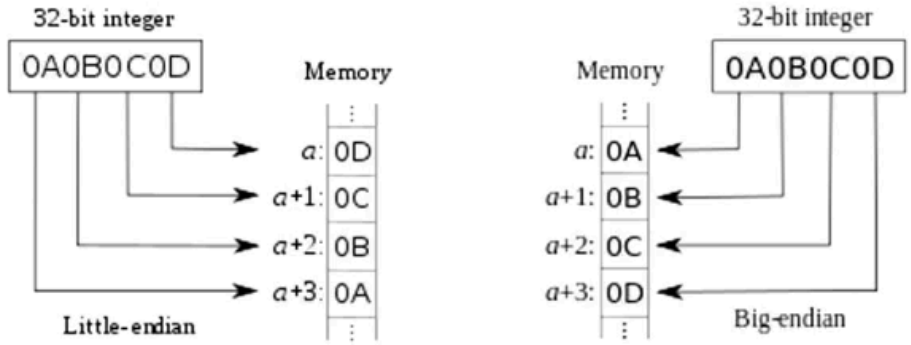
LC2K instructions

`add` (R-type instruction)	0b000	Add contents of `regA` with contents of `regB`, store results in `destReg`.
`nor` (R-type instruction)	0b001	Nor contents of `regA` with contents of `regB`, store results in `destReg`. This is a bitwise nor; each bit is treated independently.
`lw` (I-type instruction)	0b010	"Load Word"; Load `regB` from memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`sw` (I-type instruction)	0b011	"Store Word"; Store `regB` into memory. Memory address is formed by adding `offsetField` with the contents of `regA`. Behavior is defined only for memory addresses in the range [0, 65535].
`beq` (I-type instruction)	0b100	"Branch if equal" If the contents of `regA` and `regB` are the same, then branch to the address `PC+1+offsetField`, where `PC` is the address of this beq instruction.
`jalr` (J-type instruction)	0b101	"Jump and Link Register"; **First** store the value `PC+1` into `regB`, where `PC` is the address where this `jalr` instruction is defined. **Then** branch (set PC) to the address contained in `regA`. **Note** that this implies if `regA` and `regB` refer to the same register, the net effect will be jumping to `PC+1`.
`halt` (O-type instruction)	0b110	Increment the `PC` (as with all instructions), then halt the machine (let the simulator notice that the machine halted).
`noop` (O-type instruction)	0b111	"No Operation (pronounced no op)" Do nothing.

Converting instructions to machine code
Addressing modes for lw/sw instructions (big endian, little endian)

Big Endian vs. Little Endian

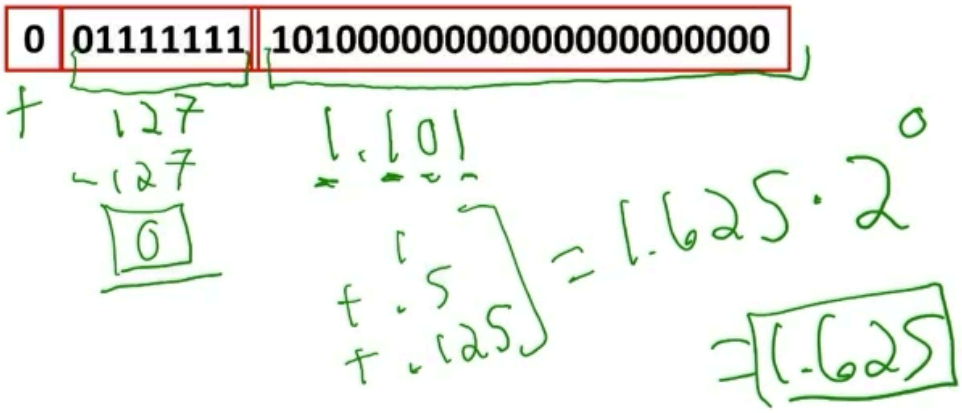
- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - The Internet is big endian, x86 is little endian, LEG and ARMv8 can switch



- In LC2K, registers are 32 bits, and in ARM, they're 64 bits

IEEE 754 Floating Point

- What is the value, in binary, of the following IEEE 754-encoded floating-point number?
- What is the value, in decimal, for the same number?



- ---

LC2K is word (32 bits / 4 bytes) addressable while ARM is byte addressable

- Sign extend if necessay

Struct alignment/padding

- Size of each data type will be provided

Minimum Datatype Sizzes

Type	Minimum size (bits)
char	8
int	16
long int	32
float	32
double	64

- In 64 bit system, pointers are 8 bytes and in a 32 bit system, pointers are 4 bytes (aka, it's the same number of bits as the system)
- Alignment rules:
  - Every primitive (aka not array/struct) data type must start at an address divisible by its size

- The first element of a struct must start at an address that is divisible by the size of the largest primitive element of the struct
- Padding must be added to the end of the struct so that the total size of the struct is a multiple of the size of the largest primitive element in the struct

Memory Layout

- How many bytes does the C data structure require (assuming a 64-bit machine)?

```
struct foo {
 double *w;
 char x;
 int y;
 char z[10];
};
```

←8 bytes
←1 byte
←4 bytes + 3 bytes for alignment
←1 * 10 bytes
←26 bytes + 6 for padding= 32 bytes

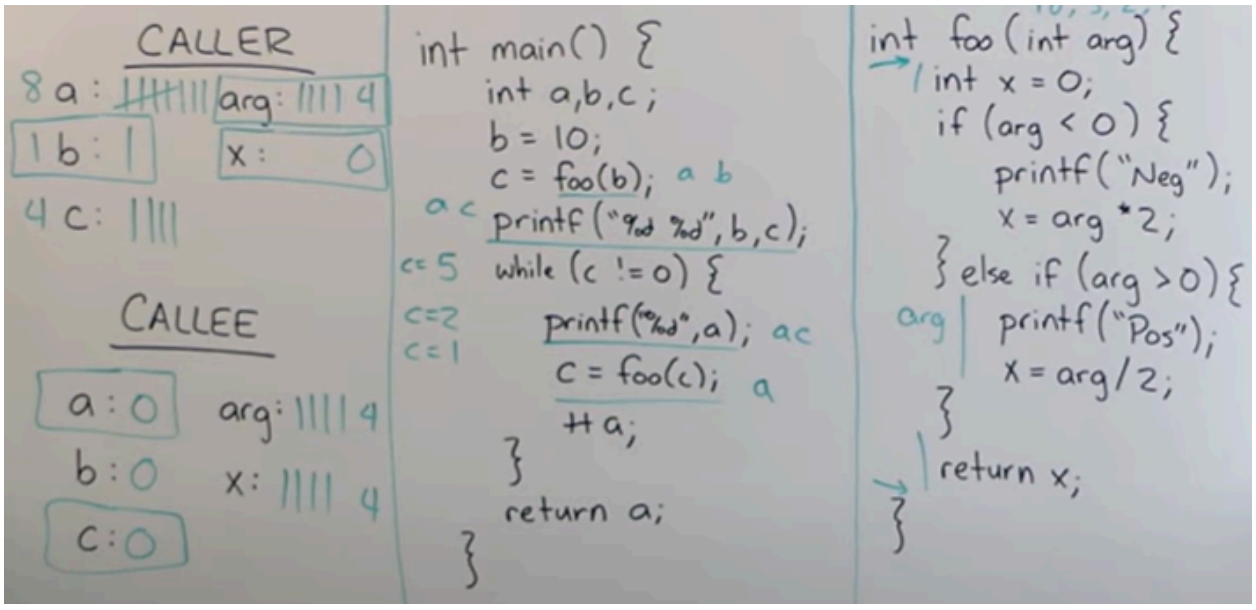
- How could this structure be rewritten to reduce memory usage?

- Note: ordering from largest to smallest or smallest to largest minimizes padding

Data organization (stack, heap, static, text)

Stack frames

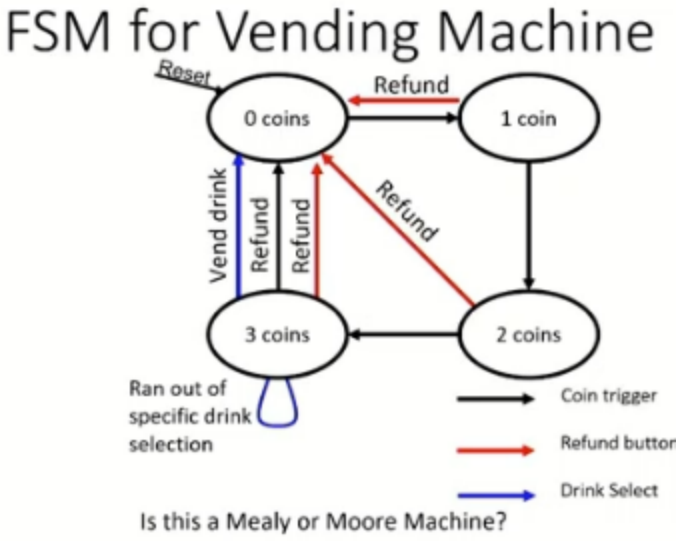
[Caller vs. Callee save]



Compiler, linker, loader

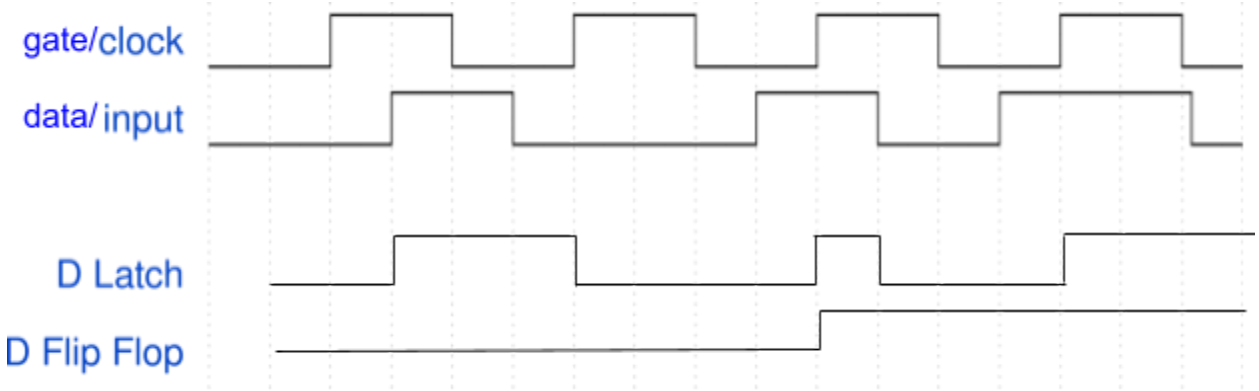
Logic gates and basic state machines

- Where S is a set of states and I is a set of inputs:
  - P(S) is a Moore machine; output depends only on current state
  - P(S, I) is a Mealy machine; output depends on current state and inputs. Example:

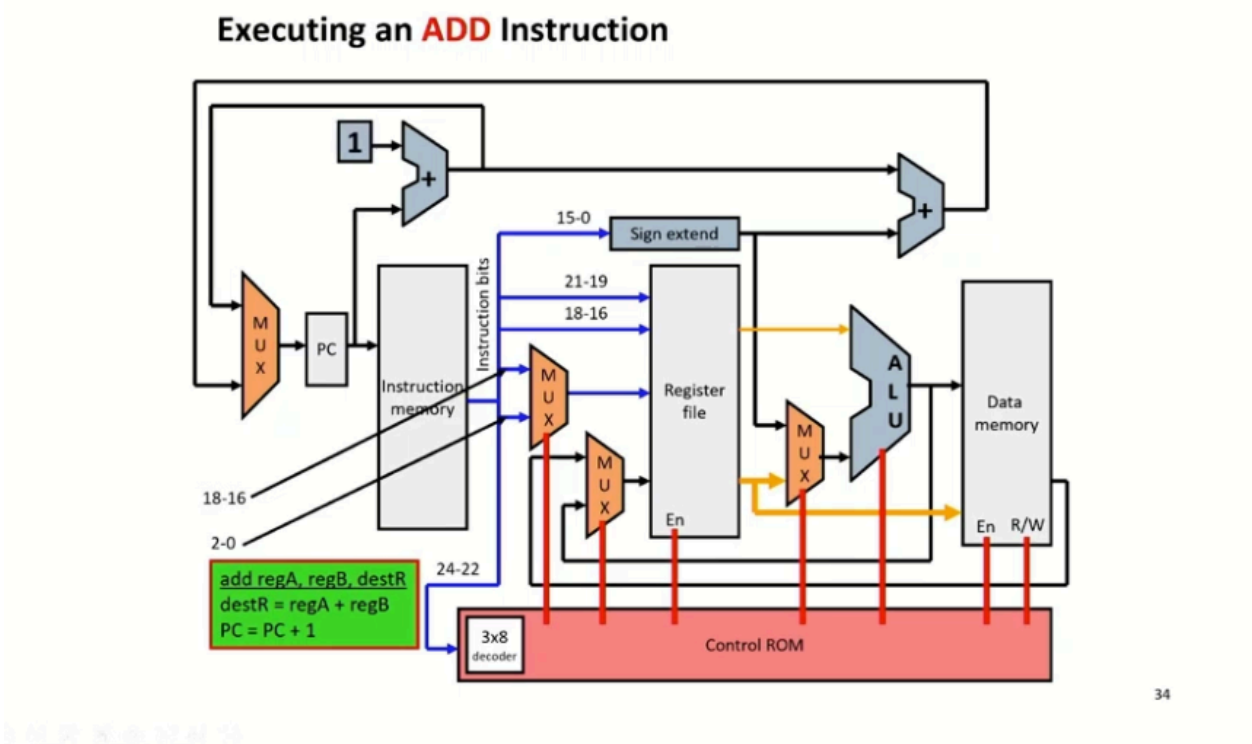


-

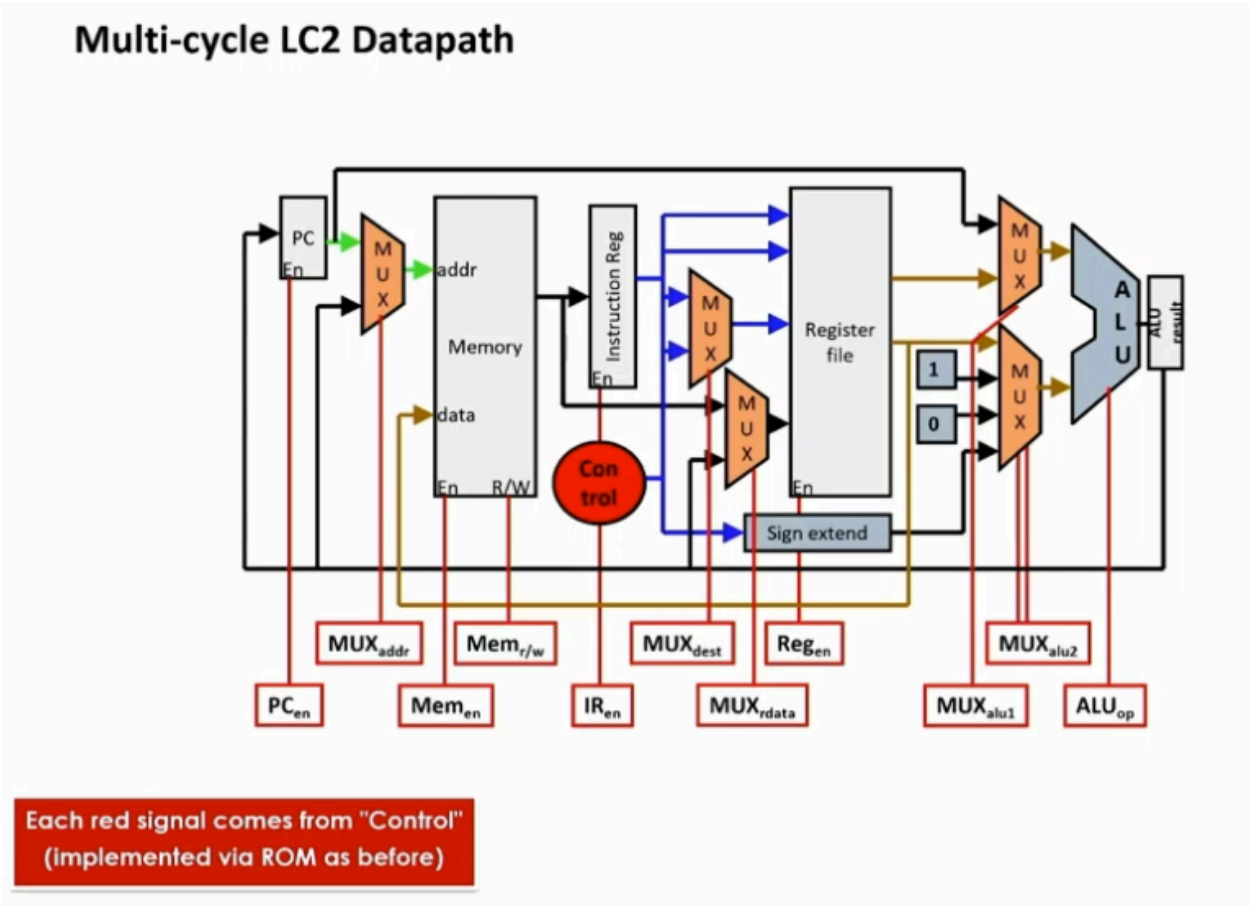
latches vs. flip flops (be able to draw diagram)



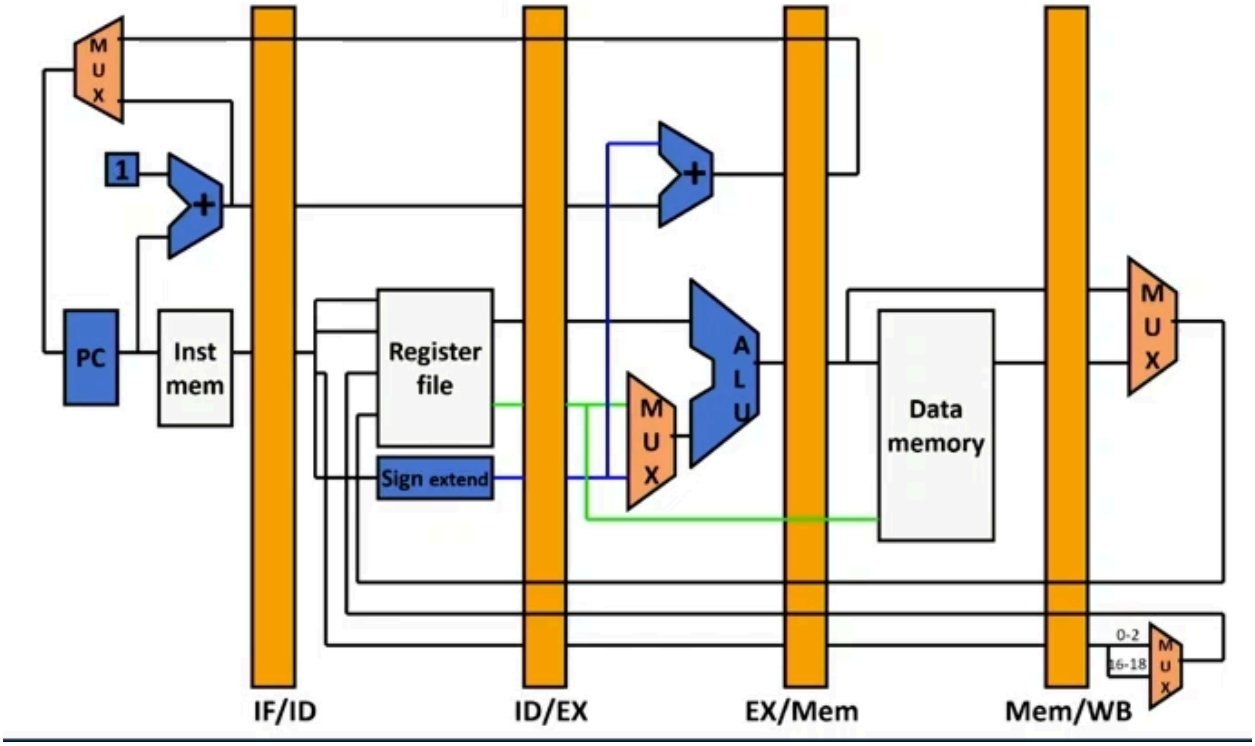
Single cycle processor



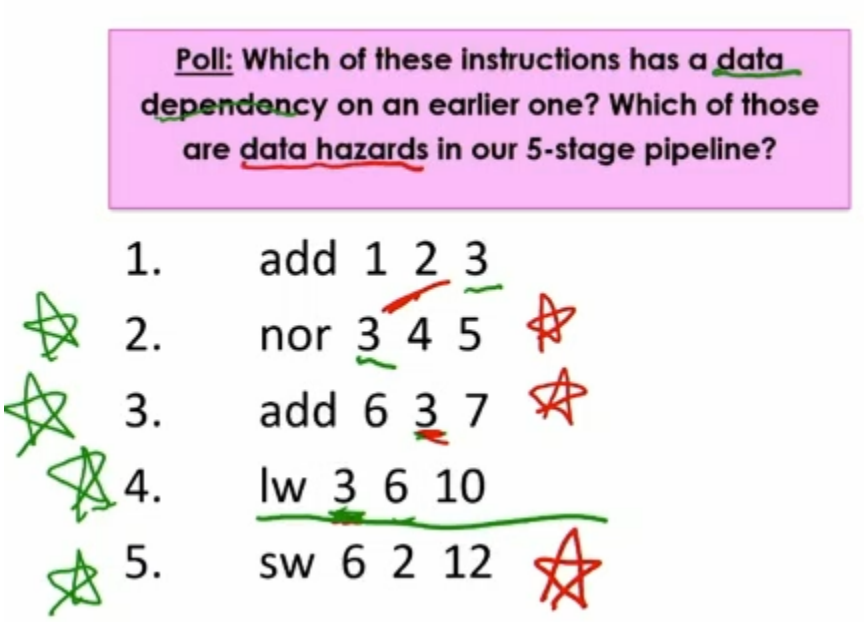
Multi-cycle processor


Multi-cycle LC2 Datapath
PC
En
M
U
X
addr
Memory
data
En
R/W
Instruction Reg
En
Con
trol
Register
file
En
Sign extend
1
0
ALU
ALU result
MUX_addr
Mem_r/w
MUX_dest
Reg_en
MUX_alu2
PC_en
Mem_en
IR_en
MUX_rdata
MUX_alu1
ALU_op
Each red signal comes from "Control"
(implemented via ROM as before)

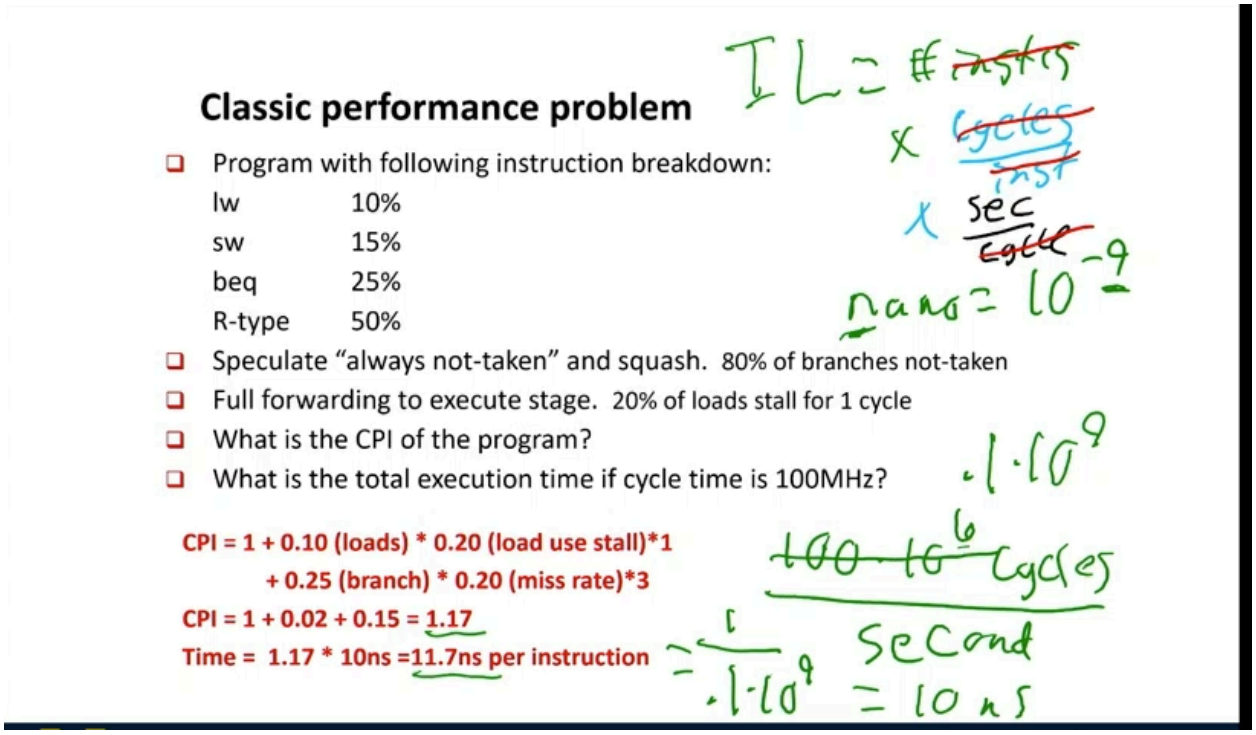
Pipelining


M
U
X
1
+
PC
Inst
mem
Register
file
Sign extend
M
U
X
+
A
L
U
Data
memory
M
U
X
0-2
16-18
M
U
X
IF/ID
ID/EX
EX/Mem
Mem/WB

- **Data hazards**: since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read an old / stale value before the correct value is written back.
- **Control hazards**: A branch instruction may change the PC, but not until stage 4. What do we fetch before that?

- Deal with data hazards by forwarding values. Eliminates need to stall at all for add/nor, but you will need to stall one cycle for a lw followed by a sw.
- Control hazards: predict that branch is not taken. Only lose 3 cycles if branch was taken; otherwise no performance lost
- Data dependency- any instruction that depends on the result of a previous instruction
- Data hazard- any data dependency where a wrong value could be obtained because of a data dependency that wasn't dealt with (within 2 of add/nor, or a sw within 2 of a lw; stall in decode stage until previous instruction writes back)
- 

Calculate performance



- Lose 3 cycles per branch taken
- Stall 1 cycle per lw followed by a data dependency

Adding new instructions

Symbol Table/ Relocation Table

- Defined global labels go in the symbol table
- Accessing a variable in a section that is different from where it is defined needs to go into the relocation table. Accessing local variables does not go into the relocation table
- Static can be local

Symbol Table & Relocation Table

File main.c

```
1: int r;
2: extern int x;
3: extern void foobar();
4: void main(int n) {
5:   reference to x
6:   reference to r
7:   foobar();
8: return; }
```

File foobar.c

```
1: int x;
2: int y;
3: void foobar() {
4:   int t;
5:   reference to x
6:   reference to y
7:   reference to t
8: return; }
```

What symbols appear in the symbol tables?

What instructions appear in the relocation tables?

Symbol Table:

`r, x, foobar, main`

Relocation Table:

`5, 6, 7`

Symbol Table:

`x, y, foobar`

Relocation Table:

`5, 6`