Measuring Performance:

CPU time = CPU clock cycles × CPU clock cycle time

clock cycle time = 1/clock rate

CPU time = CPU clock cycles / clock rate

↓ implies

To improve ~~program~~ ~~system~~ program performance:

reduce # of cycles or increase clock rate

Example: A program runs in 15 seconds on a 2GHz processor. We want to design a computer (CPU) that can run it in 10 seconds.

CPUA

CPUB

CPU A time = CPUA clock cycles / CPUA clock rate

$15 = \text{CPUA clock cycles} / 2 \times 10^9$

CPUA clock cycles = 3×10^{10} = CPUB clock cycles

$10 = 3 \times 10^{10} / \text{CPUB clock rate}$

CPUB clock rate = 3×10^9

The CPU Performance Equation -

$$\text{CPU Time} = \frac{\text{Instruction count} \times \text{cycles per instruction}}{\text{Clock rate}}$$

	Class A (4)	Class B (3)	Class C (6)
P1	450	375	200
P2	375	130	250

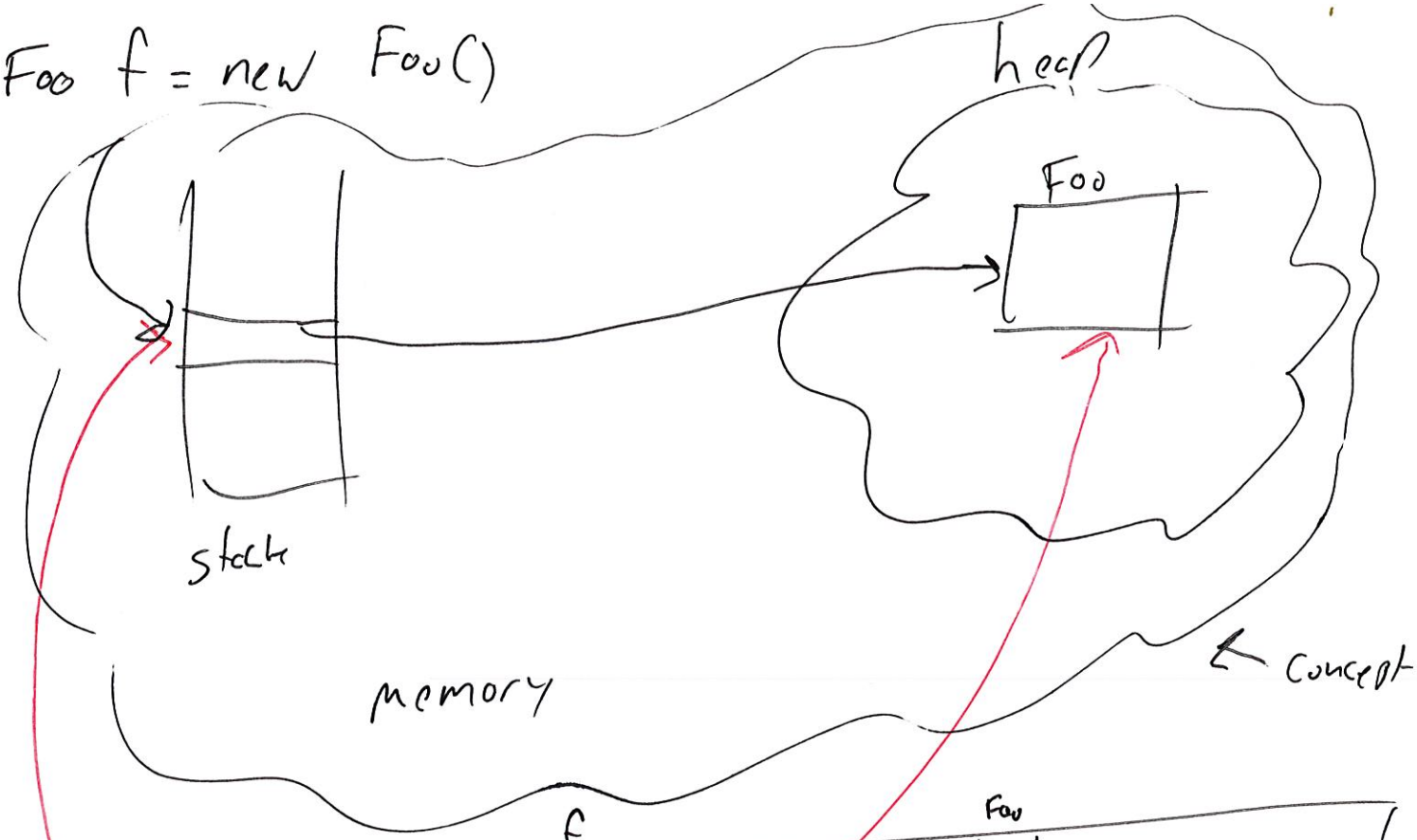
$$\text{CPU 1 time} = \frac{450*4 + 375*3 + 200*6 = 4125}{3.5\,GHz}$$

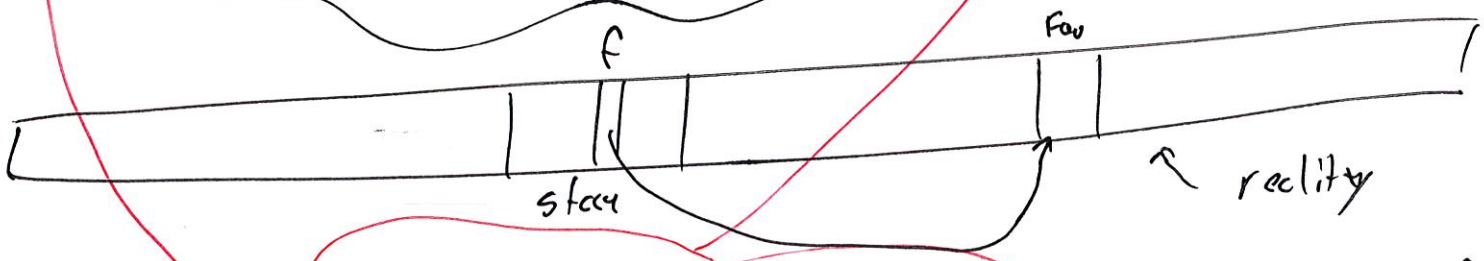
$$\text{CPU 2 time} = \frac{375*4 + 130*3 + 250*6 = 3390}{3.5\,GHz}$$

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \cancel{\frac{\text{Instructions}}{\text{Program}}} \times \cancel{\frac{\text{Cycles}}{\text{Instruction}}} \times \cancel{\frac{\text{Seconds}}{\text{cycle}}}$$

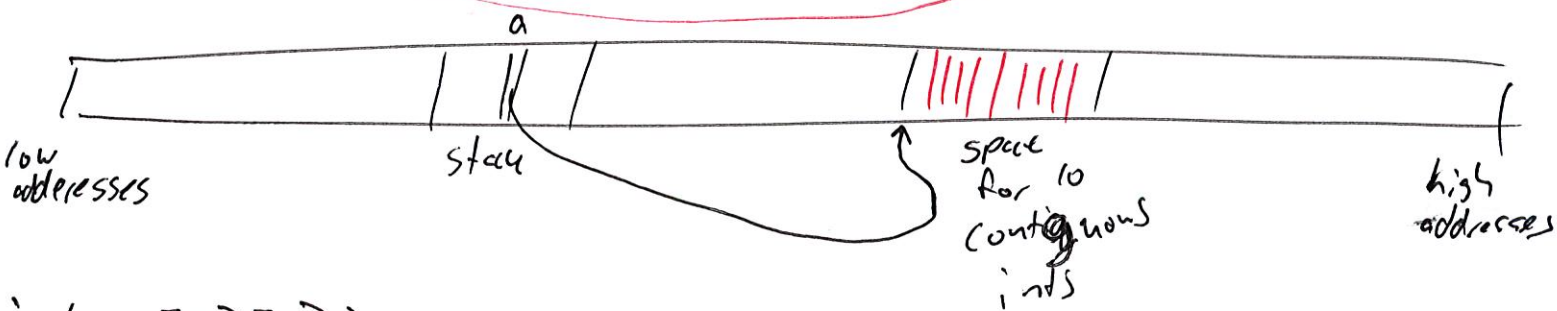
$$\left.\begin{array}{l} \text{Power} \propto \text{Voltage}^2 \times \text{frequency} \\ \text{voltage} \propto \text{frequency} \quad \text{— roughly} \end{array}\right\} \rightarrow \text{Power} \propto \text{voltage}^3$$

Foo f = new Foo()





int *a = malloc(10 * sizeof(int)); // allocates array for 10 ints



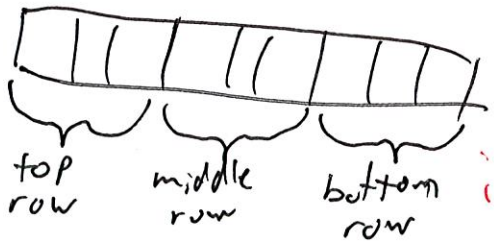
int m[3][3];

conceptually

correct

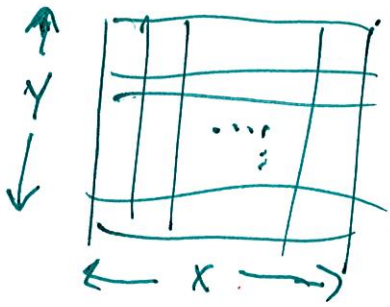
*(m[j][i]) == m + j*3 + i

in practice



in assembly: m + 4*(j*3+i)

int m[y][x]



m[j][i]

m + j * X + i

"linear index"

in C, arrays and pointers are (roughly) the same thing

m + 4 * (j * X + i)

compiler puts this here for you

Representing binary integers

Place value

$$321 = 3 \cdot 100 + 2 \cdot 10 + 1 \cdot 1$$

hundred's place, ten's place, one's place

definition of place value (in base 10)

$$d_n d_{n-1} \cdots d_2 d_1 d_0 = \sum_{i=0}^{n} 10^i \cdot d_i$$

In base 2:

$$b_n b_{n-1} \cdots b_2 b_1 b_0 = \sum_{i=0}^{n} 2^i \cdot b_i$$

Binary place value

$12_{10} = 1100_2 = 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$

$2^3 \quad 2^2 \quad 2^1 \quad 2^0$

8's 4's 2's 1's
place

$127_{10} = 1111111_2$

$-5 = -101$ ← Fine on paper, but computers don't have negative signs

Instead use most significant bit (MSB) as sign

$-5 = 1101$

↑ could be negative 5 if I specify that I have 4 bit values

$$\begin{array}{r} 15 \\ +\ 9 \\ \hline 24 \end{array} \qquad \begin{array}{r} 111 \\ 1111 \\ +\,1001 \\ \hline 11000_2 = 24_{10} \end{array}$$

16 8

$$\begin{array}{r} 15 \\ -\ 9 \\ \hline 6 \end{array} \qquad \begin{array}{r} 1111 \\ -1001 \\ \hline 0110 = 6_{10} \end{array}$$

4 2

↑ Computers don't do it this way

One's Compliment

- Negate a value by inverting all bits

$15_{10} = 01111_2$ 01001

$-9_{10} = 10110_2$ 10110

$$\begin{array}{r} 15 \\ -\ 9 \\ \hline 6 \end{array} \rightarrow \begin{array}{r} 111 \\ 01111 \\ +\,10110 \\ \hline 00101 \end{array}$$

← not the right answer what do we do with the carry?

end-around carry (add carry to one's position)

$$\begin{array}{r} 1 \\ 00101 \\ 1 \\ \hline 00110_2 = 6_{10} \end{array}$$

Why end around carry? Consider zero

$$\begin{array}{r} 00000_2 \\ +\ 11111_2 \\ \hline 11111_2 \end{array} \qquad \begin{array}{r} 0_{10} \\ -0_{10} \end{array}$$

One's complement has ±0

Two's Compliment

To negate: invert all bits, the add 1

Results in 1 more negative than positive

signed char have range [-128, 127]

when adding, end carries are discarded

$$
\begin{array}{r}
15_{10} \\
-\ 9_{10} \\
\hline
6_{10}
\end{array}
\quad
\begin{array}{r}
{\scriptstyle 1\ 1\ 1\ 1\ \ } \\
01111_2 \\
+\ 10111 \\
\hline
00110_2
\end{array}
$$

Throw away carry

$-9 \rightarrow 01001$

$\rightarrow 10110$

$\rightarrow 10111_2$

When we need binary in a program, we usually write hexadecimal or octal instead

Hexel	values	Octal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	
9	1001	
a	1010	
b	1011	
c	1100	
d	1101	
e	1110	
f	1111	

$127_{10} = 1111111_2$

$128_{10} = 10000000_2$

$73_{16} = \underbrace{\boxed{0111}}_{7}\underbrace{\boxed{0011}}_{3}{}_2$

LEGv8 assembly – Pedagogical subset of ARMv8

LEGv8 = Lessons + Extra Instructions + Garrulous Intx

~~Most~~

Register instructions: mnemonic dest, firstOpReg, secondOpReg
Immediate instructions: mnemonic dest, OperandReg, #immValue
Data instructions: mnemonic dest, [addressReg, #offset]

Registers are named X0 – X31

Example: Compile (by hand) the line of C code

$$y = x + a[15] + 7$$

Assume y is in X14, x is in X15 and a is in X16

have to multiply by 8!

load → LDUR X14, [X16, #120]
add immediate → ADDI X14, X14, #7
add → ADD X14, X14, X15

Jumping:

B – Branch, jumps unconditionally to a label
labels have the form "name:"

CBZ – Compare and Branch if zero, takes a register and a label
CBNZ – " " " not " ...

Example:

```
if (x == y)
    x++;
else
    y++;
```

Assume x -> X15, y -> X16

```
SUB X17, X15, X16     // X17 = X15 - X16
CBZ X17, increment_x
ADDI X16, X16, #1     // y++
B end
increment_x:
ADDI X15, X15, #1     // x++
end:
```

Example:

```
while (a[i])
    i++;
```

Assume a -> X21, i -> X22

0001_2

1000_2

load dest, [addr, offset]

offset imm

```
top_of_loop:
LSL X23, X22, #3
~~LDUR X23, [X21 X23]~~
ADD X23, X23, X21
LDUR X23, [X23, #0]
CBZ X23, end
ADDI X22, X22, #1
B top_of_loop
end:
```

loop body begins

loop body ends

```
MOV dest, src      // pseudo instruction
ADD dest, src, XZR
or ADDI dest, src, #0
```

a[i] in C, equivalent to *(a + i)

compiler takes address of a, calculates size of things a points to and multiplies i by that size, then adds that quantity to a

---

Procedure calls

In order to call a procedure, the program must:

1) Place the parameters ~~in~~ - In ARM, parameters go in X0 - X7
2) Transfer control - that is, jump to procedure
3) Allocate storage for saved registers ~~for~~ for automatic (local) variables
4) Perform calculation
5) place the result - Return value(s) goes in registers X0-X7
6) Return control - restore saved values & jump to caller

Registers to facilitate procedure calls:

SP (Stack pointer, X28) - Points to top of stack
FP (Frame pointer, X29) - Points to the bottom of the frame
LR (Link register, X30) - Return address for procedure

Instructions to facilitate procedure calls:

BL (branch and link). Parameter is address of procedure. Branches to procedure and stores address of current instruction (PC - program counter) in LR

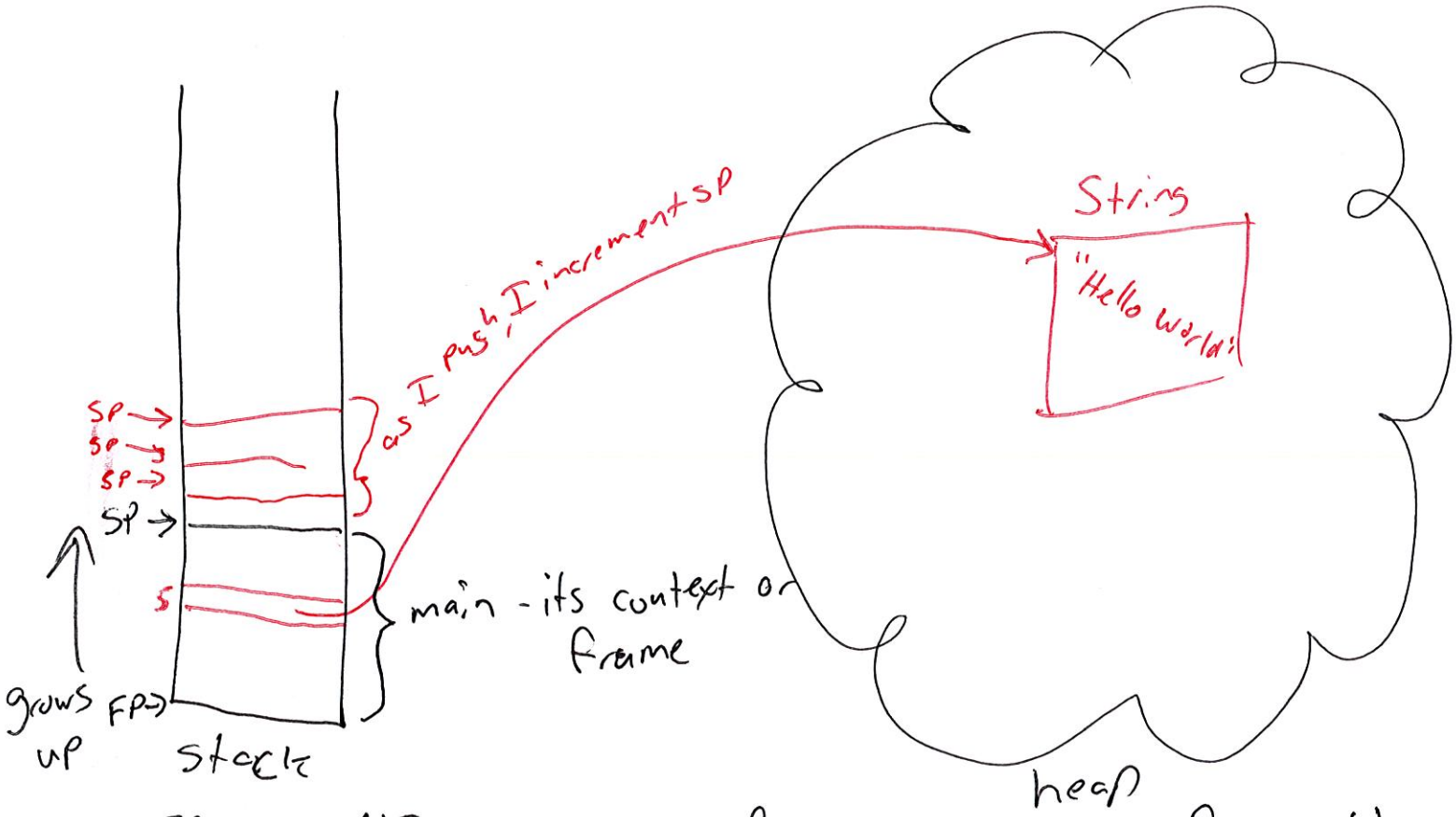
BR (Branch to register). Jumps to the value in the LR + 4 (instructions are 4 bytes so address of next instruction after procedure call)

passed as parameter

9

Assume we are in main

String s = new String("Hello World");



FP is useful because it is fixed relative to a frame, while ~~stack~~ SP moves as we push and pop. FP is optional, SP is "required"

To implement a procedure:

```
myProcedure:    // label
// save saved registers (only the ones we intend to use)
SUBI  SP, SP, #8 * n (n is the number of things we need to save)
STUR  reg, [SP, #8 * (n-1)]
        ⋮
STUR  reg, [SP, #0]  // store the n things
Procedure body
ADD   X0 reg, XZR // store return value in X0 (if needed)
// Restore saved registers
LDUR  reg, [SP, #0]
        ⋮
LDUR  reg, [SP, #8 * (n-1)]
ADDI  SP, SP, #8 * n   // restore stack pointer
BR LR
```

opposites

To call a proceedure:

```
// save temporaries that you still need to the stack (if any)
SUBI SP, SP, #8*n // n is number of items to save
STUR reg, [SP, #8*(n-1)]
        :
STUR reg, [SP, #0]
BL myproceedure
// restore saved registers (if any)
LDUR reg, [SP, #0]
        :
LDUR reg, [SP, #8*(n-1)]
ADDI SP, SP, #8*n
```

← Fill parameter registers (X0-X7)

```
long long add_one (long long i)
{
    return i + 1;
}
```

i is in X19 (a saved register)

```
ADD X0, X19, XZR
BL add_one
ADD X19, X0, XZR
```

```
add_one:
ADDI X0, X0, #1
BR LR
```

```
long long fib(long long n)
{
    if (n < 2) {
        return n;
    }
    return fib(n-1) + fib(n-2);
}
```

```
fib:
SUBI X9, X0, #2
CBZ X9, done
SUBI X9, X0, #1
CBNZ X9, body
done:
BR LR
body:      // stack manipulation
SUBI SP, SP, #24
STUR X20, [SP, #0]
STUR X19, [SP, #8]
STUR LR, [SP, #16]
// save parameters
ADD X19, X0, XZR
// set parameter for first call fib(n-1)
SUBI X0, X19, #1
BL fib
ADD X20, X0, XZR
// set parameter for second call, fib(n-2)
~~ADD X20, X0,~~
~~ADD X0, X19, XZR~~
~~SUBI X0, X0, #2~~
SUBI X0, X19, #2
BL fib
ADD X0, X0, X20 // adding result of 2 recursive calls
LDUR LR, [SP, #16]
LDUR X19, [SP, #8]
LDUR X20, [SP, #0]
ADDI SP, SP, #24
BR LR
```

Preventing Data Races

A data race occurs when two or more threads of control want to read and write the same location

LEGv8 has two instructions to handle synchronization:

LDXR - load exclusive register
STXR - store exclusive register

Given:

```
LDXR Xn, [Xm, #0]
// optionally, some other code
STXR Xp, Xq, [Xm]
```

If at any time between the LDXR and the STXR the data stored at Xm is changed STXR does not change Xm and Xq is set to 1; otherwise, the store to Xm succeeds and Xq is set to zero

An atomic exchange:

```
again:
LDXR X10, [X20, #0]
STXR X23, X9, [X20]
CBNZ X9, again
ADD X23, XZR, X10
```

Atomic exchange can be used to implement a lock

To implement a lock:

```
ADDI X11, XZR, #1
again:
LDXR X10, [X20, #0]   // X20 initialized 0, meaning no lock
                      // non-zero is locked
CBNZ X10, again
STXR X11, X9, [X20]
CBNZ X9, again
```

And to release it:

```
STUR XZR, [X20, #0]
```

Encoding Instructions - Translating from Assembly to machine language

ADD X10, X19, X23

mnemonic — Rd, Rn, Rm

10001011000	~~[illegible]~~	10111	000000	10011	01010
opcode	~~[illegible]~~ Rd	X23 Rm	shamt	X19 Rn	X10 Rd

10001011000101110000001001101010 ↔ ADD X10, X19, X23

Opcodes are an example of a prefix code. No opcode starts with another opcode

LSL X9, X19, #17

11010011011	00000	010001	10011	01001
opcode	Rm (don't care)	shamt 17	X19 Rn	X9 Rd

ADDI X25, X2R, #3134

mnemonic — Xd, Xn, imm

1001000100	110000111110	11111	11001
opcode	3134 immediate	X2R Rn	X25 Rd

3134
−2048
1086
−1024
62
32
30
16
14
8
6
4
2
2
0

5 7 3 8 2

5 3 7 2 8

3 5 2 7 8

3 2 5 7 8

2 3 5 7 8

Bubble Sort example

B ~~foo~~ done (in Fibonacci program, page 12 of class notes, done == 16)

000101 00000000 00000000000000 10000 ← wrong

opcode for b

26 bits

21 zeros

leaves 26 bits for target

23 zeros

000101 00000000000000 00000000000100 ← less wrong, but still wrong, because

opcode

All instructions are aligned on a multiple of 4 bytes. That is, lowest-order 2 bits are always zeros. Therefore, we can increase the length of the longest possible jump by a factor of 4 by discarding the lowest-order 2 bits

$10^3 \approx 2^{10}$

$2^{26} = 2^6 \cdot 2^{20}$

Still one more problem: ARM (and most (all?) architectures uses "PC-relative addressing". Branch targets are offsets applied to the PC of the branch

2nd instruction from fibonacci (Page 12)

CBZ X9, done

10110100 00000000000000000000 01001 ← correct

opcode — 19 bits pc-relative Branch target (3) — X9

Decoding instructions:

1111 1000 0000	0100 0000	00	10 0010	0001
opcode	DTP address	op	Rn	Rt
STUR	64		X17	X1

STUR X1, [X17, #64]

1110 1010 0000	1011	0000 00	01 0100	1001
	Rm	Shamt	Rn	Rd

ANDS X9, X10, X11

Rd = Rn & Rm

ADDI X9, X12, #120

STUR X3, [X19, #8]

1101 0001 0000 0000 1010 1001 1110 0101

1101 0010 1100 1111 1111 1111 1110 1110

1001 0001 0000 0000 0001 0000 1100 1001

ADDI 00000 1111 000

1111 1000 0000 0000 1000 0010 0110 0011

opcode — address — op — X19 — X3

Condition Registers

GE — SUBS
LE — SUBIS
GT
LT
EQ

SUBS X0, X1, X2

X0 = X1 - X2

GE = X1 >= X2
LE = X1 <= X2
GT = X1 > X2
LT = X1 < X2
EQ = X1 == X2

SUBS XZR, Xm, Xn

SUBS XZR, X1, X2

B.cond

B.GE dest // Branch to dest iff X1 > X2

MOV (pseudo instruction)

MOV Xn, Xm ← equal

ADD Xn, Xm, XZR

SUBI X5, X15, #42

MOVZ X14, #32767, LSL 32

static int foo = 0x

X += 3

x+=

~~1010~~

00000 1111 000

Multiplication in binary

First decimal:

```
     328
    x415
    ----
    1640
    3280
+  13120
  ------
  136120
```

```
  328
- 256
  ---
   72
  -64
    8
```

```
  415
- 256
  ---
  159
 -128
   31
```

```
      1010000000
     x 11001111 1
     ------------
      1010001000
     1010001000
    1010001000
   1010001000
  1010001000
 1010001000
1010001000
+ 1010001000
------------
10001001110111 1000
```

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

```
131072
  4096
  2048
  1042
    32
    16
+    8
------
136120
```

```
  23
   7
 161
```

```
  10111
    111
  -----
  10111
 10111
10111
-------
101 00001
```

```
  10111
    111
  -----
  10111
 10111
10111
-------
101000001
.......
```

How does the computer do it (multiplication)?

Always positive; keep track of sign

Multiplication by bit n is an n bit shift.

For example, A 64 bit multiply requires 63 adds

To optimize: Do all shifts at once (in parallel)

Add using a tree structure

32 adds, then 16, then 8, 4, 2, 1

1 2 3 4 5 6 add stages

→ 6 cycles for addition + 1 cycle for shifts

=> 7 cycles

How to do division

$17 \times 3 = \boxed{51}$

```
        14
17 ) 238
     -170
      68
      68
       0
```

```
238
128
---
110
 64
---
 46
 32
---
 14
  8
---
  6
```

```
          1110
10001 ) 11101110
        10001
        -----
         11001110
         10001
         -------
          100011 0
          100010
          ------
               0
```

2.8
5 / 14
10
40

2r4
5 / 14
10

5: 0101
14: 1110

initialization zeros

pads → width of the operation

quotient	divisor	remainder	remainder − divisor
0000	01010000	00001110	10111110 (negative)
0000	00101000	00001110	11100110
0000	00010100	00001110	11111010
0000	00001010	00001110	00000100
0001	00000101	00000100	11111111
0010			

shift to right

shift in one because previous subtraction was positive

replace w/ difference

integer division quotient

remainder

# of steps = bit width + 1

10
101 / 1110
101
0100

~~Scratch space~~ Scratch space

~~01010000 − 00001110 → invert 11110001 / 11110010 −14~~

~~01010000 + 11110010 = 101000010~~

00001110 − 01010000 → invert 10101111 +1 = 10110000

00001110 + 10110000 = 10111110

00001110 − 00010100 → 11101011 / 11101100

00001110 + 11101100 = 11111010

00001110 − 00101000 → invert 11010111 +1 = 11011000

00001110 + 11011000 = 11100110

00001110 + 11110110 = 00000100

00000100 − 00000101 → 11111010 +1 = 11111011

00000100 + 11111011 = 11111111

n % 2 − slow

n & 1 − fast

$3.75_{10} = 11.11_2 = 1.11 \times 2^0$

$= 1.111 \times 2^1$

0 10000000 111 -- 20 zeros

sign exponent

$-11.84375 = -1011.11011 \times 2^0$

34375

09375 $= -1.0111101 \times 2^3$

03125

1 10000010 01111011 -- 15 zeros

S

.5

.25

.125

.0625

.03125

IEEE 754 Floating point

highest order

lowest order

S - sign, 1 bit

exponent, 8 bits

mantissa, 23 bits

$$S \cdot -1 \; \underbrace{1.\text{mantissa}}_{\text{significand}} \times 2^{(\text{exponent} - 127)}$$

Examples:

$$2.0_{10} = 10.0_2 = 10.0_2 \times 2^0$$

$$= 1.0_2 \times 2^1$$

$$0 \; \underbrace{10000000}_{\substack{\text{8 bits} \\ \text{exponent} \\ 2^8 = 128 \; -127 = 1}} \; \underbrace{00000000000000000000000}_{\text{23 zeros}} = 2$$

$$5.0_{10} = 101.0_2 = 101.0 \times 2^0$$

$$= 1.01 \times 2^2$$

exponent $-127 = 2$

exponent $= 129$

$$\underset{\text{sign}}{0} \; \underset{\text{exponent}}{10000001} \; \underset{\text{mantissa}}{01 \cdots 21 \text{ zeros}}$$

To add floating point numbers:

* shift the significand of the number w/ the smaller exponent to the right until the ~~binary points line up~~ exponents match.
* add the significands
* normalize
* round

5_{10} = 0 10000001 01 ··· 21 zeros 101

3.75_{10} = 0 10000000 111 ··· 20 zeros

5 → 1.01 1.01

3.75 → 1.111 shift → + 0.1111

10.0011 $\times 2^2$

1.00011 $\times 2^3$ ← renormalize

0 10000010 00011 ··· 18 zeros

$$\begin{array}{r} 5 \\ +\ 3.75 \\ \hline 8.75 \end{array}$$

1000.11 $\times 2^0$

1.00011 $\times 2^3$

$$\begin{array}{r} +\ 11.84375 \\ +\ 2.0 \\ \hline 13.84375 \end{array}$$

0 10000010 01111011 ··· 15 zeros

0 10000000 23 zeros

$$\begin{array}{r} 1.01111011 \times 2^3 \\ +\ 0.01000000 \\ \hline 1.10111011 \end{array}$$

0 10000010 10111011 ··· 15 zeros

To multiply floating point numbers:

* add the exponents.
* multiply the significands
* normalize
* round
* sign

$$\begin{array}{r} 4.7 \\ 3 \\ \hline 14.1 \end{array}$$

		s	exp	
5_{10}	=	0	10000001	01 ... 21 zeros
3.75_{10}	=	0	10000000	111 ... 20 zeros

$$\begin{array}{r} 1.010 \\ \times\ 1.111 \\ \hline \end{array}$$

$$\begin{array}{r} 1.111 \times 2^{1} \\ \times 1.010 \times 2^{2} \\ \hline 1111 \\ +\ 1111 \\ \hline 10.01011 \times 2^{3} \end{array}$$

$$1.001011 \times 2^{4}$$

$$\begin{array}{r} 3.75 \\ 5 \\ \hline 18.75 \end{array}$$

$$10010.11$$

positive 2^{4}

0 10000010 001011 ... 17 zeros

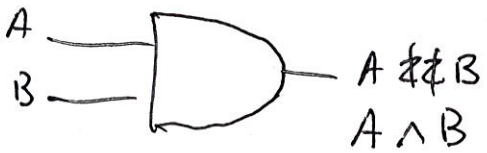
Digital Logic

Combinational Logic: Logic whose value (result) depends only on its input values

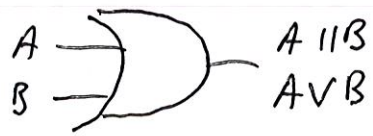
Sequential Logic: Logic whose value depends on the values of its inputs and on internal state (that is, state that was configured on earlier inputs)

Some ~~compact~~ combinational structures:

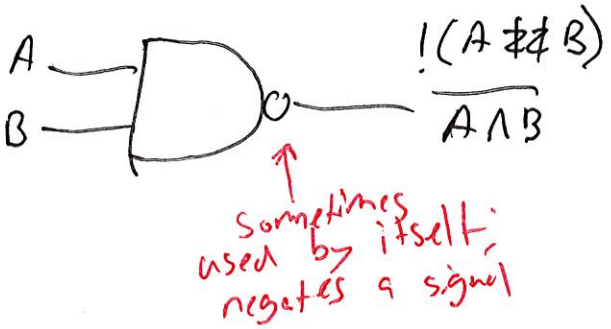
AND Gate



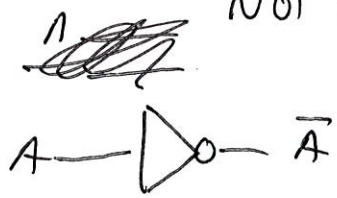
OR Gate



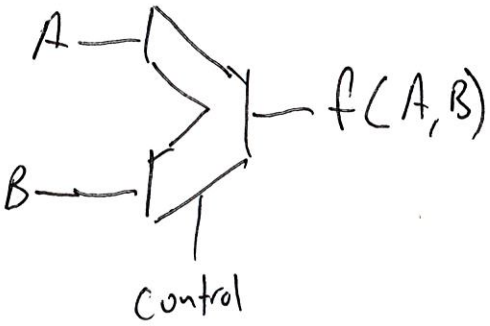
NAND Gate



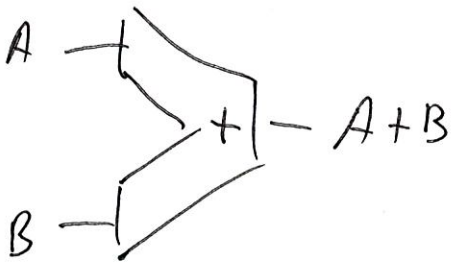
Not Gate



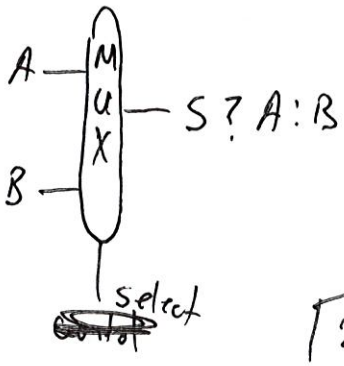
ALU



Fixed Function

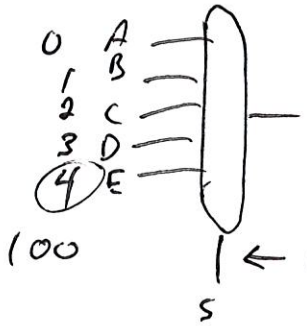


mux
multiplexer



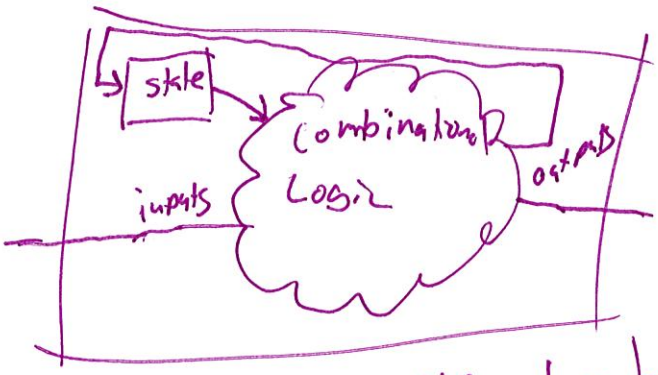
Combinational Structures (cont.)

Multiplexers can have any number of inputs, need enough bits in the selector to uniquely select.

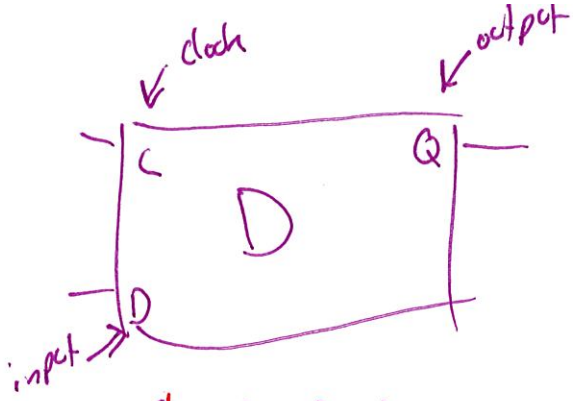


← Bus must be 3 bits wide to select among 5 inputs

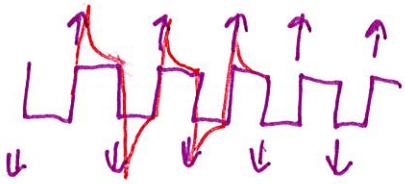
Sequential Logic



Everything is this box happens at the same time
For some period of time the output values are in a state of flux because they are some undefined combination of the inputs, the saved ~~state~~ state, and the old inputs. Over time, the circuit will stabelize to a "steady state" where output is correct.

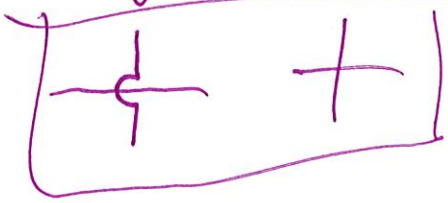


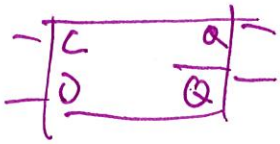
D-flipflop - stores a single bit, the value of D when the clock goes up; Q is the last stored value

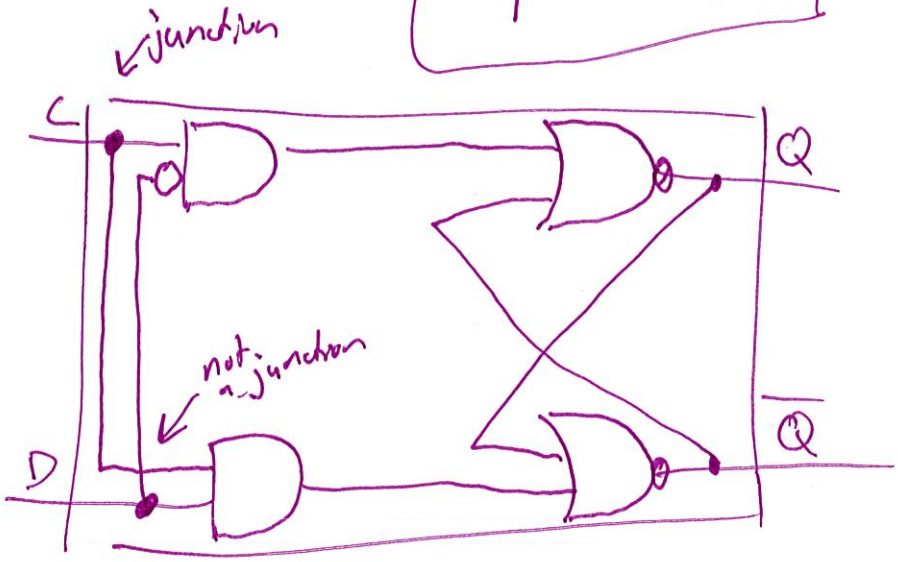


← clock signal

don't do this anymore

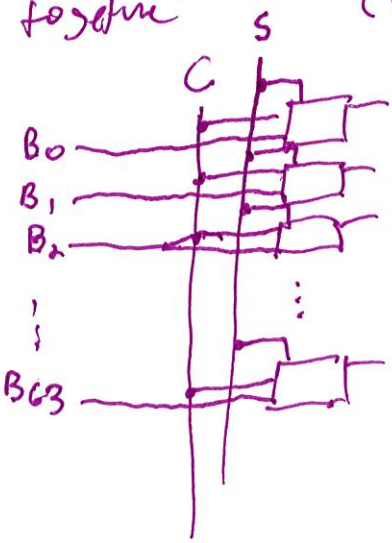


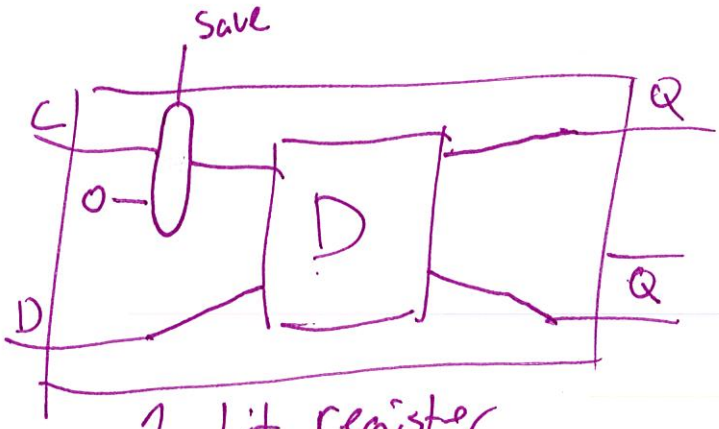




stores 1 bit

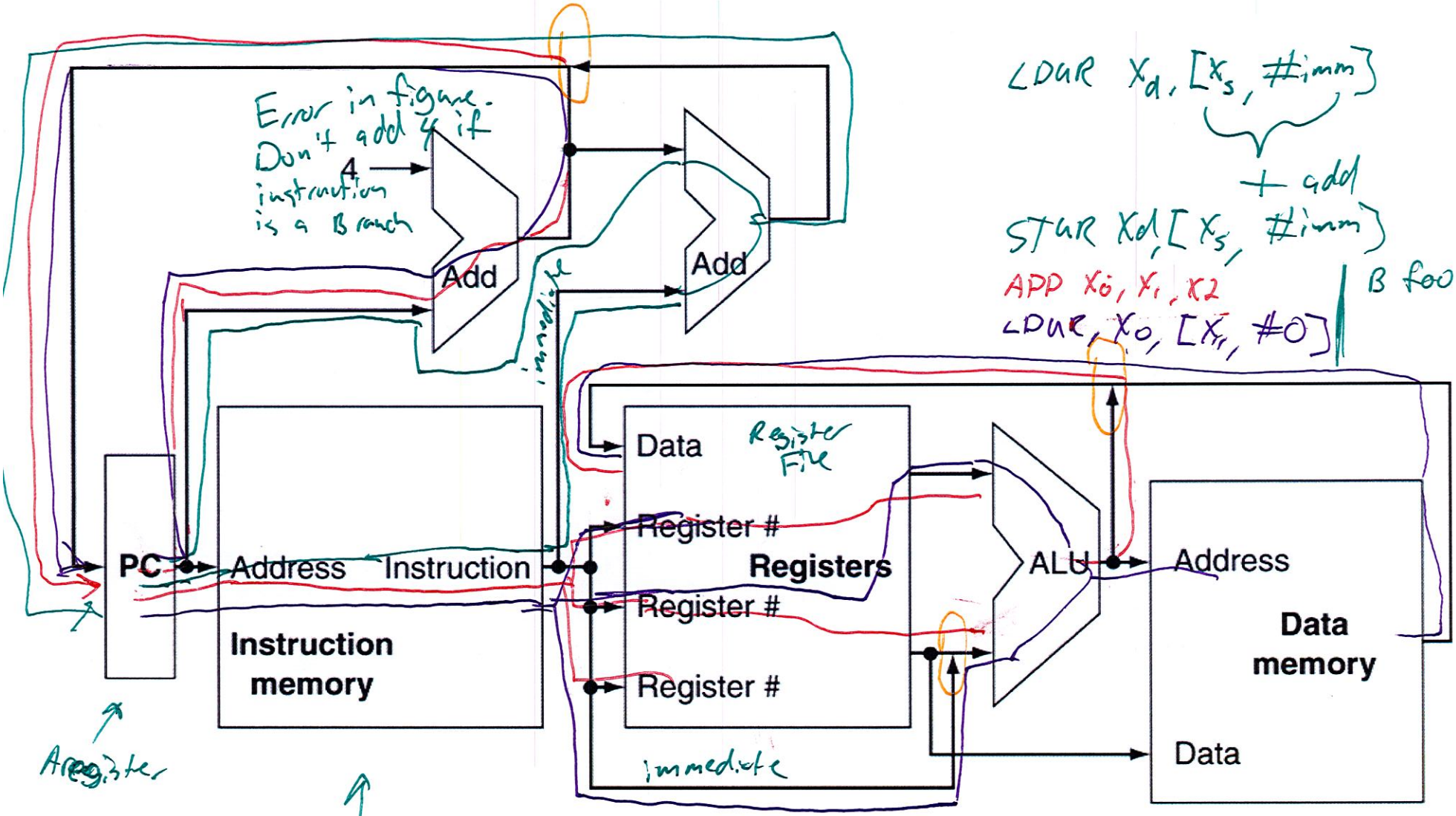
To make an, e.g., 64-bit register, put 64 D-flipflops together (one-bit registers)





1 bit register

This whole processor is a single sequential circuit. Outputs are not valid until the entire thing reaches steady state. Clock frequency must be set accordingly.



LDUR X_d, [X_s, #imm]

+ add

STUR X_d, [X_s, #imm]

APP X_0, X_1, X_2

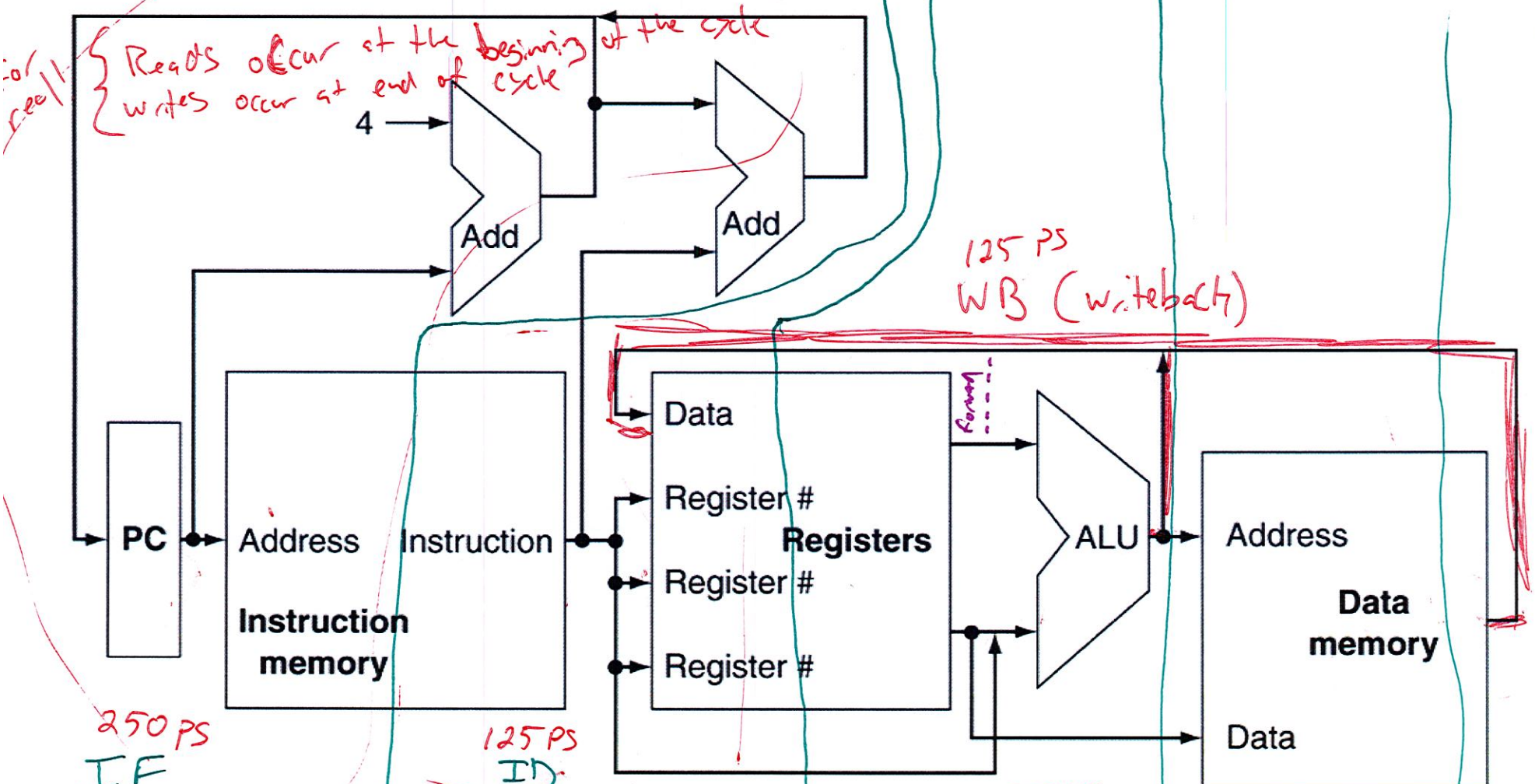
LDUR, X_0, [X_1, #0]

B foo

Seperate memory paths for instructions and data

Traditional 5-stage pipeline

1 ns → 1 GHz = 1/1ns



125 PS
WB (writeback)

250 PS
IF
(Instruction Fetch)

125 PS
ID
~~DEC~~
(decode)
(Instruction decode)

ExE 250 PS
(execute)

MEM 250 PS
(memory)

4 GHz
1/250 PS

unpipelined - Clock speed is 1/time to steady state for the entire processor
pipelined - clock speed is 1/time to steady state for the highest latency stage

1 ADD X9, X10, X4
2 ADD X10, X9, X12

Instruction # \ Cycle # →	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EXE	MEM	WB					
2		IF	ID	EXE	MEM	~~ID~~ WB				

value of X9 not available yet! so this schedule is not valid

	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EXE	MEM	WB					
2		IF	ID	—	—	EXE	MEM	WB		

Bubble

Implement "Data forwarding" to get result of EXE sent into EXE in next instruction for next cycle

Called a "Data hazard"

	1	2	3	4	5	6	7
1	IF	ID	EXE	MEM	WB		
2		IF	ID	EXE	MEM	WB	

1 LDUR X10, [X0, #0]
2 ADD X9, X10, X11

	1	2	3	4	5	6	7	8	9
1	IF	ID	EXE	MEM	WB				
2		IF	ID	—	—	EXE	MEM	WB	

No forwarding

	1	2	3	4	5	6	7	8	9
1	IF	ID	EXE	MEM	WB				
2		IF	ID	—	EXE	MEM	WB		

with forwarding from MEM to EXE

X10's new value not available until end of cycle

to do this without a bubble, we'd need a Delorian and a flux capicitor

X = Y + Z

W = Z + V

X	Y	Z	V	W
#0	#8	#16	#24	#32
X_0	X_1	X_2	X_3	X_4

2 LDUR X_1, [SP, #8]

2 LDUR X_2, [SP, #16]

3 ADD X_0, X_1, X_2

4 LDUR X_3, [SP, #24]

5 ADD X_4, X_2, X_3

We can forward to here

nothing to forward yet. Need value from inst 2 in same cycle

	1	2	3	4	5	6	7	8	9	10
1	IF	ID	EX	MEM	WB					
2		IF	ID	EX	MEM	WB	EX	MEM	WB	
3			IF	ID	–	–	EX	mem	WB	
4				~~IF~~ IF	~~ID~~ ID	~~ID~~	EX	MEM	WB	
5						IF	ID	~~MEM~~	EX	MEM WB

one cycle of babble

1 LDUR X_1, [SP, #8]

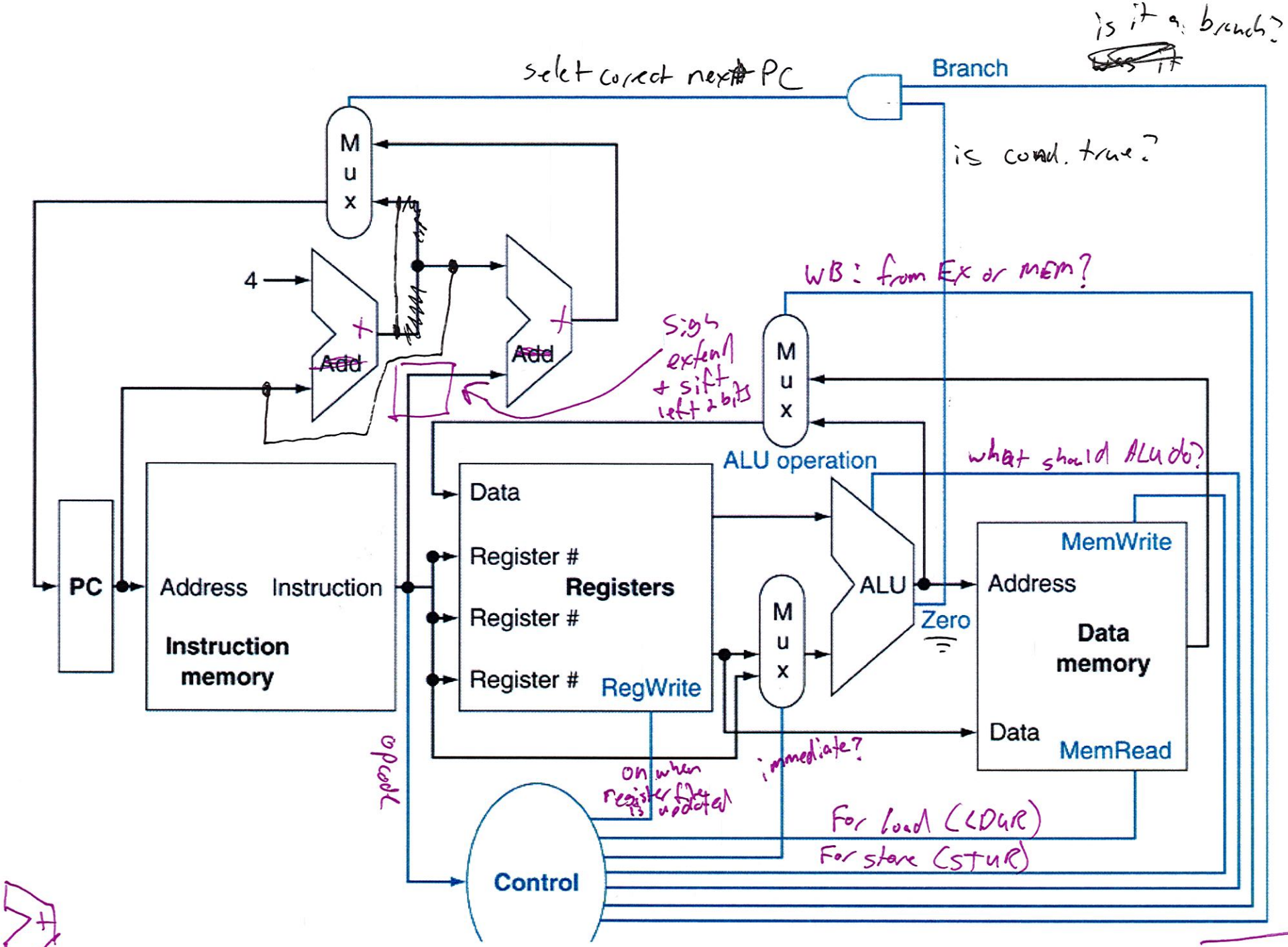
2 LDUR X_2, [SP, #16]

4 LDUR X_3, [SP, #24]

3 ADD X_0, X_1, X_2

5 ADD X_4, X_2, X_3

IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
			IF	ID	EX	MEM WB


is it a branch?
select correct next PC
Branch
is cond. true?
M u x
4
Add
Add
WB: from Ex or MEM?
Sign extend + shift left 2 bits
M u x
ALU operation
what should ALU do?
PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
RegWrite
M u x
ALU
Zero
MemWrite
Address
Data memory
Data
MemRead
opcode
on when register file updated
immediate?
For load (LDUR)
For store (STUR)
Control