Textbook Notes 2.1.7

Intro to boolean Algebra

$\sim$ = not, & = and, | = or, ^ = XOR

we can use these bitwise operators on bit strings.

$a = [a_{w-1}, a_{w-2}, \dots, a_0]$ & $b = [b_{w-1}, b_{w-2}, \dots, b_0]$

such that $a \& b = [a_{w-1} \& b_{w-1}, a_{w-2} \& b_{w-2}, \dots, a_0 \& b_0]$

we can use bit strings to represent finite sets.

eg. $A \subseteq \{0, 1, \dots, w-1\}$ w/ $[a_{w-1}, \dots, a_1, a_0]$

where $a_i = 1$ iff $i \in A$

so $A = \{0, 3, 5, 6\} \Rightarrow a = [\boxed{0}1101001]$ ← why start with extra zero?

$B = \{0, 2, 4, 6\} \Rightarrow b = [01010101]$

$\Rightarrow A \cap B = a \& b = [01000001]$

4.2.1 Logic Gates in Hardware Control language

Processors designed ~~use~~ notated w/ Verilog / VHDL (like languages)

- room for an XML language for ~~processor~~ Hardware design?
- Basically

CS51 Textbook Notes

2.1.1

Trick for powers of 2

$x = 2048 = 2^{11} \Rightarrow n = 11 = \overset{\uparrow}{3} + 4 * 2$

$i + 4j = n$

$i = 0 \Rightarrow$ leading 1

$i = 1 \Rightarrow$ leading 2 $\qquad j = \#$ of zeros

$i = 2 \Rightarrow$ leading 4

$i = 3 \Rightarrow$ leading 8

$2048 = 2^{11} \qquad 11 = 3 + 4 * 2$

$\Rightarrow$ leading 8 $\quad \Rightarrow$ 2 0s $\quad \Rightarrow$ 0x800

To convert ~~hex~~ dec to Hex,
divide x by 16 $\Rightarrow$ quotient q, remainder r.
hex(r) = least sig digit then repeat on Q
$\Rightarrow$

$314{,}156 = 19634 \cdot 16 + 12 \quad (C)$

$19634 = 1227 \cdot 16 + 2 \quad (2)$

$1227 = 76 \cdot 16 + 11 \quad (B)$

$76 = 4 \cdot 16 + 12 \quad (C)$

$4 = 0 \cdot 16 + 4 \quad (4)$

$\Rightarrow \boxed{0x4CB2C}$

Conversely,
~~0x4CB2C~~ $0x7AF = 7 * 16^2 + 10 * 16^1 + 15$

2.2.2 unsigned ints

$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$ (Binary to unsigned of length w

What is the max ~~# of~~ ~~unsigned~~ unsigned ints we can represent w/ w-len bit string

$UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$

$\Rightarrow UMAX_4 = \sum_{i=0}^{3} 2^i = B2U_4([1111]) = 2^4 - 1 = 15$

Define $B2U_w$ as mapping

$B2U_w : \{0,1\}^w \longrightarrow \{0, \ldots, 2^w - 1\}$

a bijection - unique value to each bit vector of w each int. b/t 0 and 2^w-1 has 1 uniqu bin. composition.

2.2.3 Two's-complement Encodings

Define most significant bit to have negative weight.

$\Rightarrow$

$B2T_w(\vec{x}) \doteq -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$

Range:

$TMin_w \doteq -2^{w-1}$

$TMax_w \doteq \Sigma_{i=0}^{w-2} 2^i = 2^{w-1} - 1$

so $B2T_4 : \{0,1\}^w \rightarrow \{-8, 7\}$. Also a bijection

Note. $|TMin| = |TMax| + 1$

$|UMax| = 2TMax + 1$

-1 in two's-comp. $=$ Umax $=$ 0xFFFF

Textbook notes ctd.

Almost all machines require 2's comp. representation of signed ints.
These limits set in file <limits.h> in C library
corresponding to $Tmax_w$ = INT_MAX, $Tmin_w$ = INT_MIN,
$UMax_w$ = UINT_MAX.

$\}$ For portability in C, check out <stdint.h> w/ intN_t...
& set of macros w/ INTN_MIN, ... etc. (p.63)

Two alt. representations:
1. Ones' complement:
Same as 2's, except most sig bit has weight $-(2^{w-1}-1)$
instead of -2^{w-1} ie $B2O_w(\vec{x}) = -x_{w-1}(2^{w-1}-1) + \sum_{i=0}^{w-2} x_i 2^i$
2. Sign-magnitude.
Most sig bit = sign to determine whether other bits get positive or negative weight ie. $B2S_w(\vec{x}) = (-1)^{x_{w-1}} \left(\sum_{i=0}^{w-2} x_i 2^i \right)$
Both ⟹ multiple representations of 0.
Ones' comp. = outdated
we use sign-magnitude w/ floating point #'s

2.2.4

in C, you can cast b/t dif. numeric types.

- Doing so ⟹ same bits, but reads diff.

Mathematical relationship:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$

vs.

$$U2T_w(u) = \begin{cases} u, & u < 2^{w-1} \\ u - 2^w, & u \geq 2^{w-1} \end{cases}$$

Textbook Notes

2.3.1

- Strange things can happen w/ finite computer arithmetic
- Lisp supports infinite

<u>Unsigned Arithmetic:</u>

consider x, y s.t. $0 \le x, y \le 2^w - 1$ (aka w-bit #'s)

BUT $x + y$ could be ~~2~~ $= 2^{w+1} - 2 \Rightarrow w+1$-bit

~~Lets~~ Thus we do modular arithmetic.

Drop the leading digit (AKA, if $x+y > 2^w$, $x+y \Rightarrow x+y-2^w$

eg. $x = 9 = 1001$, $y = 12 = 1100$

$x + y = 21 = 10101$

$\Rightarrow$ drop leading 1 $\Rightarrow 0101 = 5$

$21 - 2^4 = 21 - 8 = 5 = (x+y) \% 2^w$

AKA it overflows.

full int. result cannot fit w/in the word size limits of the data type when operands sum to more than 2^w

Define $+^u_w$ operation

$$x +^u_w y = \begin{cases} x + y, & x+y < 2^w \\ x+y-2^w, & 2^w \le x+y < 2^{w+1} \end{cases}$$

~~Df~~ ~~$s<x$~~

Let's say $s \doteq x +^u_w y$

if $s < x$, we know overflow occurred.

This forms abelian groups.
- commutative & associative, has identity element 0

consider set of w-bit unsigned #'s w/ $+^u_w$

$\Rightarrow -^u_w x +^u_w x = 0$, when $x = 0$, it's clearly 0

$$\Rightarrow -^u_w x = \begin{cases} x & x = 0 \\ 2^w - x, & x > 0 \end{cases} \quad \text{for } 0 \le x < 2^w$$

(They must add to 2^w)

$\uparrow$ additive inverse of x

eg $5 = 0101 \Rightarrow 16 - 5 = 11$

$11 +^u_4 5 = 0$

Two's Compliment Addition (2.3.2)

- Given x, y in $-2^{w-1} \le x, y \le 2^{w-1} - 1$

$-2^w \le x + y \le 2^w - 2$ (could require w+1 bits to represent)

- Works the same exact way w/ bits as unsigned sum, ~~but~~ $\Rightarrow$ to weird results

$$\Rightarrow x +^t_w y \doteq U2T_w(T2U_w(x) +^u_w T2U_w(y))$$
$$= U2T_w((x+y) \bmod 2^w)$$

$-2^{w-1} \le x, y \le 2^{w-1} - 1$

$$\Rightarrow x +^t_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{positive overflow} \\ x + y, & -2^{w-1} \le x + y \le 2^{w-1} & \text{normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{negative overflow} \end{cases}$$

$2^{w-1} = 8$

eg 4-bit 2's comp addition:

$(-8) + (-5) = -13 \ (< -2^{w-1}) \Rightarrow -13 + 16 = 3$

$[1000] + [1011] = [10011] \Rightarrow [0011]$ ← cutting off a 1 = adding 2^w

$5 + 5 = 10 \ (\ge 2^{w-1}) \Rightarrow 10 - 2^w = 10 - 16 = -6$

$[0101]\ [0101]\ [01010] \Rightarrow [1010]$ ← cutting off a zero to a 1 ⇒ subtracting 2^w (cause the positive 8 → neg. 8)

~~HW~~ Textbook Notes 2.3.2 ctd.

Note, if

x & y are negative, but $x +^t_w y > 0$, negative overflow

x & y are postive, but $x +^t_w y < 0$, positive overflo

2.3.3 Two's Compliment Negation

Every x in $-2^{w-1} \le x < 2^{w-1}$ has an additive inverse of $-x$. Except -2^{w-1} (since 2^{w-1} can't be represented), so -2^{w-1} is its own inverse

'Cause $-2^{w-1} +^t_w -2^{w-1} \Rightarrow -2^{w-1} + -2^{w-1} = -2^w \Rightarrow -2^w + 2^w = 0$

$$-^t_w x \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

2.3.4. Unsigned Multiplication:

for $0 \le x, y \le 2^w - 1$,

$x * y$ could be up to $2^{2w} - 2^{w+1} + 1 \Rightarrow$ could take $2w$ bits.

unsigned multiplication takes the lower orde w-bits

$\Rightarrow$ the product modulo 2^w

$$x *^u_w y = (x * y) \bmod 2^w$$

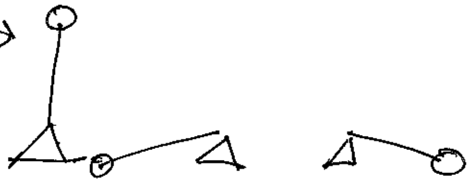
Textbook Notes 6.1

RAM:

Static RAM (SRAM) = fast & more expensive than dynamic RAM (DRAM)

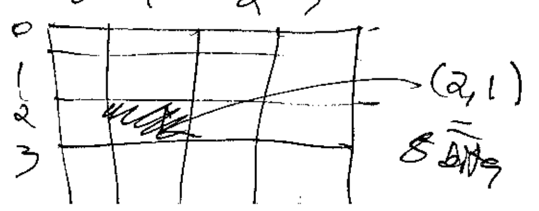
SRAM = for cache memories (A few MBs)
DRAM = for main memory & frame buffer of graphics system (hundreds+ MBs)

- Static RAM
- Stores each bit in a "bistable" memory cell (~~[illegible]~~ 6-transistor circuit)
Always goes to one of its 2 stable voltage states.
(middle state is "metastable" →



It will retain its value indefinitely as long as it's kept powered. Even w/ disturbance

- Dynamic Ram
Stores each bit on a capacitor. Can be dense.
- very sensitive to disturbance.
- Memory system must periodically refresh every bit of memory by reading & rewriting it.
- usually in supercells (eg ~~[illegible]~~ w/ 8 bits per supercell)
a 16 X 8 DRAM chip ⇒



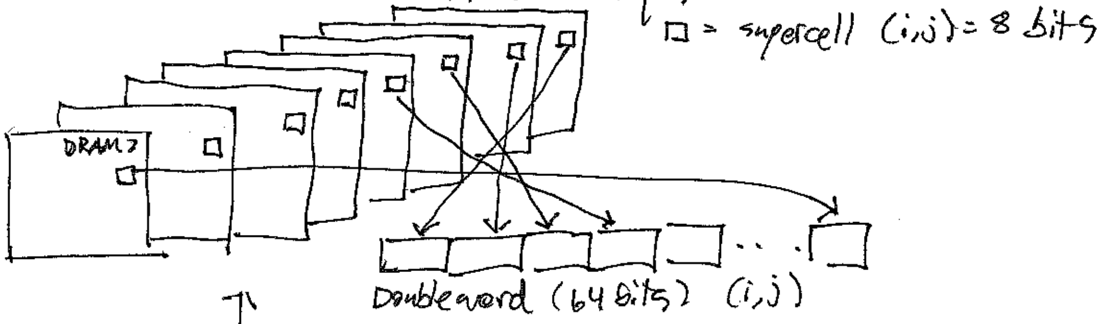
The way it works: (reading) ↙ RAS (row access strobe)

1. Memory controller sends row i to DRAM chip (which stores row i in an "internal row buffer")
2. Memory controller sends column j to DRAM chip ← CAS (which sends back 8 bits in (i, j).)

- It's a 2D array to reduce # of address pins needed.

- Memory Module:

Multiple DRAM chips in parallel, eg 64MB memory module of 8 8M X 8 DRAM chips



Duall Input Memory Module (64 bits) (DIMM)

- Non-volatile Memory

ROMs (read only memory). Don't lose data when turned off.

PROM = Programmable ROMs = ~~[illegible]~~ can be written 1X

EPROM = Erasable PROM = can be written ≈ 1000 x

EEPROM = Electrically EPROM = can be written 10^5 times

Flash memory = common now. like EEPROM

Firmware = programs stored in ROM devices.
(eg. booting, eg. input output in GPUs).

Textbook Notes 4.0 - 4.1.3

ISA = Instruction-set Architechture
= The instructions supported by a processor & their byte-level encodings.

⇨ progs. compiled for Intel IA32 vs IBM Power PC vs. ARM machines won't work on the others

Machine-level programs must access the programmer-visible state (what instructions read & Modify:
- Program registers,
- Condition Codes (CC)
- PC (program counter) (address of instruction being execlited)
- Stat (program status)
- DMEM: Memory

Prog. Registers: each stores a word (32 bits)
- %EAX, %ecx, %edx, %ebx, %esi, %edi, %esp, %ebp

%esp ↓ Stack Pointer w/ push, pop, call, return commands

Textbook Notes 4.1.2-4

Y86 Instructions:

word = 4 bytes of data

- IA32 movl ⇒ 4 parts
  - irmovl, rrmovl, mrmovl, and rmmovl
  - destination goes second
- seven jump instructions: jmp, jle, jl, je, jne, jge, & jg
- Six conditional move instructions
  - same format as r-r move instruct. rrmovl, but destination updated on conditional
  - cmovle, cmovl, cmove, cmovne, cmovge, cmovg
- Call instruction pushes the return address on the stack & jumps to the destination address.
- ret instruction returns from a call.
- pushl & popl implement push & pop (as w/ IA32)
- halt instruction stops instruction execution

Instruction Encoding:

- each = b/t 1 & 6 bytes.
- 1st byte = 2 nibbles. First a high-order "code" nibble, then a low-order "function" nibble.
- Next = sometimes a register-specifier byte (rA=Source, rB=dest.)
- If you only need 1 register specified, the other set to 0xF
- some require 4-byte constant word.
  - Can be immediate data (irmovl), displacement (rmmovl / mrmovl) & the destination of branches & call's

eg. rmmovl %esp, 0x12345(%edx)
4042 00 01 23 45 ← But, little-endian
4042 45 23 01 00

15 = 11~~0~~1 = F
00 00 00 0F
30f3 F0 00 00 00

Y86 Exceptions

Stat = status code describing state of executing program
Possible values:

1	AOK	Normal op.
2.	HLT	halt instruction encountered
3.	ADR	Invalid address encountered
4.	INS	Invalid instruction encountered

- In Y86, the program just stops, but Ideally, we'd have an instruction handler.

Operand Specs

Immediate	\$IMM	Imm	immediate
Reg.	E_a	$R[E_a]$	reg.
Mem	Imm	$M[Imm]$	absolute
mem	(E_a)	$M[R[E_a]]$	indirect
Mem	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement

Textbook Notes 3.2.0 - 3.2.1

Program Encoding:

To compile a C program on an IA32 machine,

gcc -O1 -o p p1.c p2.c

$\overset{\uparrow}{}$ level of optimization   -c $\Rightarrow$ compile & assemble

$\Rightarrow$ 1. C preprocessor includes files & expands MACROS

2. Compiler $\Rightarrow$ assembly-code, p1.s & p2.s
3. Assembler $\Rightarrow$ object-code (bin. instructs, but global vars not filled in)
4. linker merges these 2 obj. code files $\Rightarrow$ executable p

executable code = second form of machine code - what's accepted by the processor.

Machine-Level Code:

Two important Abstractions from machine implementation:

1. ISA $\Rightarrow$ Sequential processor state & instructions
2. Virtual Addresses

$\Rightarrow$ a memory model appearing as a very large byte array

In actuality, it's multiple hardware memories & OS software

There are no types in machine level code

PC, registers, CC, & float registers abstracted in C

While an IA32 processor has 32-bit addresses (4 Gigs), a program usually gets only a few megs, The OS manages this virtual address space & translates virtual $\rightarrow$ physical addresses in processor memory

GDB is great for assembling tools

Generating the actual executable requires running a linker on the set of obj. code files (one of which must contain main).

Note the diff. b/t the addresses in a linked exec & an obj. code file.
- The linker uses real addresses, not the virtual.

Textbook 3.4.0 - 3.4.2

Accessing Information:

In IA32 (X86):

8 - Registers

31		15	0
%eax	%ax	%ah	%al
ecx	cx	ch	cl
edx	dx	dh	dl
ebx	bx	bh	bl
esi	si		
edi	di		

+ esp = stack pointer
ebp = frame pointer

there are strange conventions for working w/ first 3 vs 2nd 3.

Operand Specifiers:

3 types,
i. immediate = \$-5 or \$0x1F, etc,
ii. Registers
iii. memory = accessed by a computed (AKA effective) address.

For arrays & structured elems, we have great ~~acess~~ addressing modes.
eg. scaled index
$Imm(E_b, E_i, s) \Rightarrow M[Imm + R[E_b] + R[E_i] * s]$
E_b = base register, E_i = index register, $s = 1, 2, 4,$ or 8
Imm = offset.

eg indexed
$(E_b, E_i) = M[R[E_b] + R[E_i]]$

Data movement Instructions

In IA32 movb, movw, movl (same op, but 1, 2 & 4 bytes).
- moves ~~data~~ Immediate, register, or memory data to a register or memory location.
- Can't have both source & dest. be memory locations.
- Also movs (signed expansion of little ⇒ big)
  eg. 1100 ⇒ 1111 1100 & 0100 ⇒ 0000 0100
  & movz zero expansion of small ⇒ big
  eg 1100 ⇒ 00001100

eg
```
%dh=CD , %eax = 98765432
movb %dh, %al        ⇒  %eax=987654CD
movsbl %dh, %eax     ⇒  %eax = FFFFFFCD
movzbl %dh, %eax     ⇒  %eax = 000000CD
```

```
pushl %ebp                  popl %eax
  =                           =
subl $4, %esp               movl (%esp), %eax
movl %ebp, (%esp)           addl $4, %esp
```

we can access items on stack w/o popping them by
eg movl 4(%esp), %edx
gets the second long item on stack into %edx

Textbook Notes 4.1.4 - 4.1.5 (& ch. 3)

Y86 Exceptions

- 4 ~~possible~~ status codes "Stat"
  1. AOK (normal)
  2. HLT (halt instruction)
  3. ADR (attempted to read from or write to invalid memory address)
  4. INS (invalid instruction code)

4.1.5 Y86 ~~Programs~~

Q: what's the difference b/t a base pointer & a stack pointer

Chapter 3:

- with compiled C, ignore the assembler directives (i.e. lines starting w/ '.', like .file "simple.c")

Data Formats:

"word" = 16-bits (2-bytes)
"double word" = 32-bits (4-bytes)
"quad-word" = 64-bits

- In C, most data types = double words
  - int, long int, char* (pointer).
- char data type = byte

-floating-point #'s get weird.
1. single-precision vals = 4-bytes = float
2. double-precision vals = 8-bytes = double
3. extended-precision vals = 10-bytes = long double

~~Data movement Instructions:~~
~~-mov in X86~~
~~mov ⇒ movb (1-byte), movw (2-bytes), movl (4-bytes)~~
~~-Source operand~~

Data movement Example:
-In C,
• int x = *xp;
⇒ read value stored in location designated by xp and store it as variable x. This = dereferencing.
• *xp = y;
⇒ write value of y at location designated by xp

eg. C Code | Assembly

```
int exchange(int *xp, int y)   | (#xp @%ebp+8, y @%ebp+12)
{                              |   movl 8(%ebp), %edx
  int x = *xp;                 | (# copying value at address %edx, to %eax, its the return val.)
                               |   movl (%edx), %eax
  *xp = y;                     |   movl 12(%ebp), %ecx
  return x;                    |   movl %ecx, (%edx)
}
```

```
⇒ int a = 4;
  int b = exchange(&a, 3);
  printf("a = %d, b = %d \n", a, b);      ⇒  a = 3, b = 4
```

Textbook Notes 3.5 - 3.6.2

x86 Arithmetic & Logical Operations:

- Each instruction class has 3 variants, OPb, OPw, & OPl (byte, long, word)

- Load Effective Address (leal
  a variant of the movl instruction.
  Instead of reading from memory, it copies the effective address to the destination (a register
  - leal S, D $D \leftarrow \&S$
- used for pointers & tricky arithmetic operations

- Unary & Binary Ops

• Unary:	INC	D →	D+1
	DEC	D →	D-1
	NEG	D →	-D
	NOT	D →	~D
• Binary:	ADD	S, D →	D= ~~S+D~~ D+S
	SUB	S, D →	D= ~~S-D~~ D-S
	IMUL	S, D →	D=D*S
	XOR	S, D →	D = D^S
	OR	S, D →	D = D|S
	AND	S, D →	D = D&S

In x86, source can be IMM, Mem, or Reg
Destination must be Mem or Reg
—But cannot both be Mem locations.

- Shift OPs

SAL	K, D → D = D << K	} don't wrap
SHL	K, D → D = D << K	
SAR	K, D → $D = D >>_A K$	} wraps w/ signed bit
SHR	K, D → $D = D >>_L K$	} doesn't wrap

K: single byte immediate OR in byte reg. %CL

fills

- These ~~OPs~~ OPs work for signed & unsigned #s

- Special Arithmetic OPs

imull	S → R[%edx]:R[%eax] ← S × R[%eax]
mull	S → R[%edx]:R[%eax] ← S × R[%eax]
cltd	→ R[%edx]:R[%eax] ← SignExtend(R[%eax])
idivl	
divl	

- These get weird quick, but they're not in Y86

Textbook Notes 3.6

- execution order of a set of machine-code instructions can be altered w/ a jump instruction.
- Condition Codes (single-bit registers)

  CF = Carry flag (detecting overflow for unsigned ops)
  ZF = Zero flag
  SF = Sign flag
  OF = Overflow flag (Two's compliment)

- In X86,
  - Cmp [b/w/l]$^{S_2, S_1}$ ⇒ $S_1 - S_2$ (temporary)
  - Test [b/w/l] S_2, S_1 ⇒ $S_1 \& S_2$ (temporary)
    - Don't set the registers, only the CC's
- use for ~~testing~~ questioning a val.
  testl %eax, %eax ⇒ see if %eax is zero, neg, or pos.
  OR one of operands is a mask to test only certain bits

- Accessing the Condition Codes
  ⇒ set [XX] instructions. in X86
  See this section if necessary for x86 programming...

Textbook Notes

Sequential Y86 Implementation

- On every clock cycle, SEQ performs all steps required for 1 complete instruction.
- 6 processing stages:

1. Fetch:

- read bytes of inst from memory using PC as memory address.
- Extracts the two 4-bit pieces of instr. specifier byte ⇒ icode (= instr. code) & ifun (= instr. function)
- Possibly fetches a register specifier byte ⇒ rA & rB (either or both)
- Possibly fetches a 4-byte Constant word valC
- Computes valP to be address of instr. following current one. ⇒ valP = PC + len(Instr.)

2. Decode:

- Reads up to two operands from register file ⇒ valA & valB (sometimes reading %esp)

3. Execute:

- ALU performs op, computes effective addr. of a mem ref, or incr./decr. the stack pointer. ⇒ valE
- Condition codes & branch condition (ifun) set.

4. Memory:

- read/write memory ⇒ valM

5. write back:

- writes up to two results to register file

6. PC update: - PC set to addr. of next instr.

- this loops infinitely

Trace this through:

1. OPl , rrmovl , irmovl (reg. only)
2. rmmovl, mrmovl (mem.)
3. pushl, popl (stack)
4. jXX, call, ret (control)

SEQ Hardware Structure

- mapping the 6 processing steps → the datapath

Fig 4.22

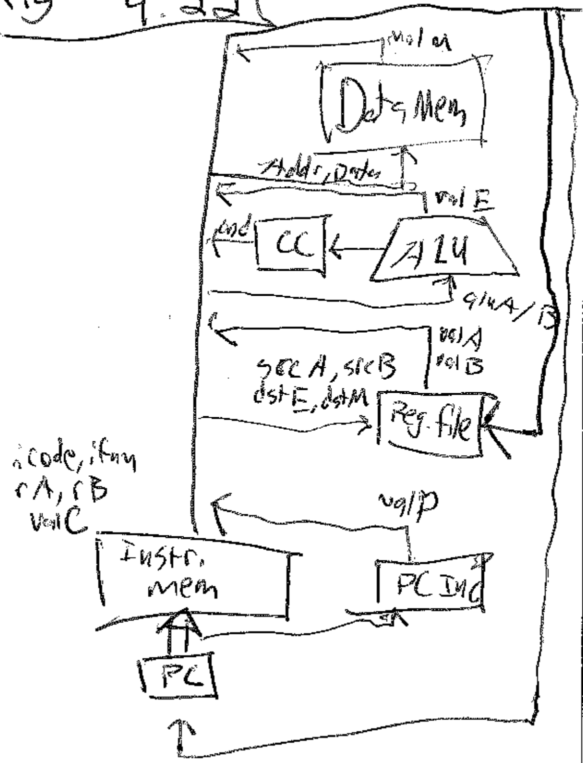
5. Fig write back

4. memor

3. execute

2. Decode

1. Fetch:



Textbook Notes 4.3.3 SEQ Timing

- A single clock transition triggers a flow of combinatorial logic to do a <u>whole</u> instruction
- Two forms of mem devices in SEQ
  1. clocked registers (the PC ~~&~~ CC)
  2. RAM (the reg. file, the instruction memory, the data memory)
- we treat reading from mem. as combinational
  ⇒ requires no sequencing or control (and we only read IMem)
  ⇒
- Only 4 hardware units requiring control over sequencing
  1. The PC
  2. the CC reg.
  3. The reg. file
  4. The data mem.
- A single clock triggers writing vals to RAM & loading into regs
- Even though we do all the oops (the steps of processing) simultaneously, we get the same effect.
- 'Cause the processor never needs to read back the state updated by an instruction in order to complete its processing

- Note, each clock cycle begins w/ the ^{4}state elems set according to the previous instruction (up)
- Signals propagate through the combinational logic creating new values for the state elems (down)
- these are uploaded on the next (up)

Textbook Notes ~~Sec~~ 4.4

Principles of Pipelining:

- The task divided into series of discrete stages
- like a cafeteria (one customer doesn't go through the line @ a time) (or carwash).
- Increases the throughput of the system.
  # customers served / unit time
- slightly increases Latency: time for one customer.

Computational Pipelines:

- customers = instructions
- we measure circuit delays in picoseconds (10^{-12}s)

$$\text{Throughput} = \frac{1 \text{ instruct.}}{x \text{ picosecs}} \cdot \frac{1000 \text{ picos}}{1 \text{ nano-sec}} = \text{GIPS}$$

(AKA Giga-instructions per sec). (Billions of instr./sec)
x = latency. (reciprocal of the throughput)
- increase in latency comes from added pipeline registers

A detailed Look at Pipeline Operation:

- Implemented by a rising-edge register-clock set up.
  ⇒ changing clock speed ⇏ fucking shit up
  but too fast ⇒ disaster

Limitations of Pipelining:

- non-uniform stage delays
  ⇒ slowest stage (longest delay) = lynchpin
  - decreases efficiency

- Diminishing Returns of deep pipelining:
  - Doubling # of partitions != doubling throughput 'cause time to update reg.s ⇒ limiting factor
  - Modern Architecture = deep pipelining (15+ stages) to maximize processor clock rate.

  ⇒ circuit designers must minimize delay on pipeline registers
  ⇒ chip designer must ensure clock distribution networ changes clock at <u>exact</u> same time

<u>Pipelining a System w/ Feedback:</u>

- what about

```
irmovl $50, %eax
addl %eax, %ebx
```

- need end-val of %eax for starting addl ⇒ data dependency

OR

```
loop:   subl %edx, %ebx
        jne targ
        . . .
targ:   halt
```

- Need control flags to tell to go w/ jne ⇒ control dependency
- ~~Depending~~ If not dealt w/, this could change program behavior (which is unacceptable).

Textbook Notes 2.1

Data sizes: (in 32-bit C)

- char = 1, int = 4, char * = 4, float = 4
- short int = 2, long int = 4, long long int = 8
- double = 8
- make programs portable. Make it insensitive to the exact sizes of diff. data types (type casting...)
  eg. you can't always stor a pointer as an int on 64-bit machines.

Addressing & Byte Ordering:

- int X ⇒ &X = 0x100 and ~~ends~~ next at 0x104
- little v. big endian (from Jonathan Swift, 1726)
  - An issue when bin. data communicated b/t dif. machines
    ⇒ established byte ordering conventions & conversions
- disassemblers = executable → instr. seq.
- OR w/ type casting (gets tricky)
  - you circumvent normal type systems

Representing Strings:

- Array of chars terminated w/ the null char
  - encoded w/ ASCII (only good for english)
  - use Unicode for more flexibility.
  - a-z = 0x61 through 0x7A

Representing Code:

- instruction codings are different on diff. machine types (even w/ identical processors).
- Bin code almost never portable.

Bit-level ops in C: (~, ^, &, |)

- Can apply bit-wise op.s in C to "integral datatypes" (AKA char or int w/ or w/o short, long, long long, or unsigned)
  eg ~0x41, 0x69 & 0x55, 0x69 | 0x55
  x = 0001   y = 0011
  y = 0001 ^ 0011 = 0010
  x = 0001 ^ 0010 = 0011
  y = ~~0010~~ 0011 ^ 0010 = 0001
- ⇒ masking
  - x & 0xFF ⇒ only least significant byte of x
  - ~0 ⇒ ~~##~~ mask of all 1s

Logical operators: (!, ||, &&)

- Any non-zero bytestring ⇒ 0x01
  eg. x = 0x66, y = 0x39
  ⇒ x && y = 0x1, x || y = 0x1, !x || !y = 0x00

Shift Ops:

- x << k ⇒ x shifted left (no wrapping) k bits
- x<<k<<j = ((x<<k)<<j)
- x>>k ⇒ x shifted right k bits (logical = no wrapping) (arithmetic = fills w/ k repetitions of most sig. bit).

Textbook Notes ~~Ch~~ 3.2-5

Machine level code:

- in IA 32, PC = %eip
  - instructions 1-15 bytes long

- Leal = load effective address
  ~~Moves~~ coppies a memory address to a register
  - AkA the C "&" operator

  leal S, D ⇒ D ← & S

  %edx has value x

  eg leal 7(%edx, %edx, 4), %eax

  ⇒ %eax ← leal [(7 + x + x*4)]

  %eax ← 5x + 7

- ↑ also good for compact arithmetic