ECE 222

Number Systems Review: Sept 10, 2018

- Carry bit ≠ overflow (usigned)
- if a, b n bits long, and b, then a+b can be rep. in n+1 bits.
  - ↳ a+b? 2n bits.
  - ↳ need $\lceil \log_2 k \rceil$ more bits.
- Ranges:
  - ↳ Sign bit: $[-(2^{n-1}-1), 2^{n-1}-1]$
  - ↳ 1's comp: $[-(2^{n-1}-1), 2^{n-1}-1]$
  - ↳ 2's comp: $[-2^{n-1}, 2^{n-1}-1]$
  - ↳ usigned: $[0, 2^n-1] \Rightarrow$ usigned($\sim A$) = 2^n - usigned(A)
    - ↑ 2's.

Signed Arithmetic:

- if a, b have dif. signs, no overflow possible (for addition!)
  - ↳ same sign but answer dif sign? overflow

Example: Sept 12, 2018

1) $-2 + -6 = -8 \Rightarrow 2^n - U(2) + 2^n - U(6)$

$= 2^n + 2^n - 8$

8

-8

```
  1 1
  1110
+ 1010
-------
1 1000
```

↑ ignore carry for signed.

Extension:

Add 1's for negative, 0's for positive. (same dig. as sign)

Sept 14, 2018

Computers:

General purpose or application specific.

Types:

1) PC's — usual
2) Workstations - powerful PC's for industry
3) Enterprise - servers, mainframes
4) Supercomputers - weather, crypto
5) Embedded - controlling systems (MP4, ABS) ← often not visible. *popular
6) Cloud - Geo - distributed. connected via internet.

Programmable Computer:

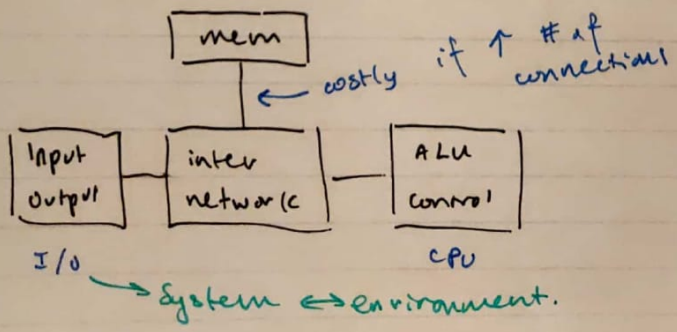
1) Store instr's
2) Exec instr's
3) conditional exec. } May be mechanical (gears)

Refer to slides for computer history. ⇒ ENIAC: nuclear bombs

Digital Computer:

- Electronic circuits for programmable comp.
- Fast, binary ← can have "noise" w voltage values
- Functional Units:
  1) Input
  2) output
  3) Memory
  4) ALU
  5) control unit.

Von Neumann Architecture:



Memory Systems:

Sept 15, 2018

linearly addressable array of cells.

→ 1 cell = 1 digit.

→ byte addressable.

Need $\log_2(\text{mem\_size})$ bits (for address)

10 bits → addr
clk
mem en
r/w

RAM 1024×8 (1KB) ↔ data (to be stored)

2048×32

2k × 4 bytes = 8KB.

← need 11 addr bits

Sec. Memory: Disks, flash.
→ large, slow

Prim. Memory: RAM
→ Med, fast

Cache: on/off chip (hardware managed)
→ small, fast
(1 for instr, 1 for data) (faster b/c they use transistors > capacitors. but, expensive.)

Registers: on chip. } Processors use this. (fast).

CPU:

Central control unit and ALU (+ registers)

ALU's: Logical/arithmetic operands

Control Unit: GET/Execute instructions from memory.

Bus:

Connection of wires used by functional units.

→ write signal to bus = bus driving

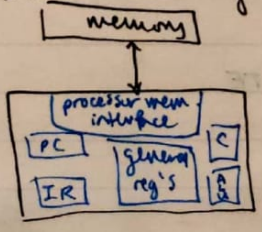
→ conflict if multiple driving at same time.

→ control unit enforces this.

Connecting CPU → memory:

Sep 18, 2018

- Program Counter, Instruction reg (current instr.), general purpose reg, control, ALU



Types of Registers:

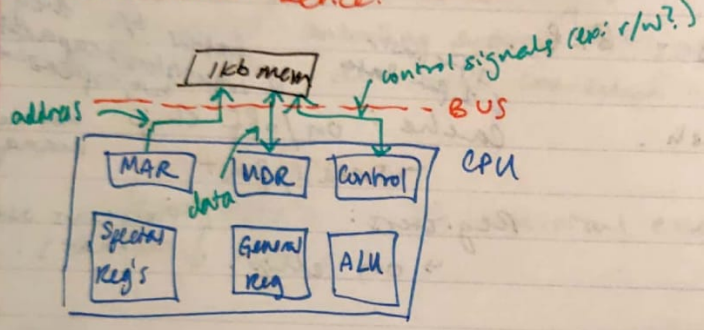
Memory data reg – temp. store data. write: data to be written. read: data that was read
Memory address reg – temp. store address, regardless of operation.

1) MAR ← PC
2) READ
3) IR ← MDR

} "Fetch"

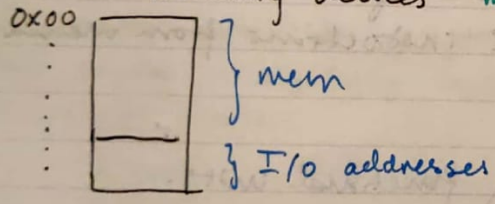
4) PC += V ← data length.

Connected Mem. Device:



Connected I/O Device:

Treat as memory devices "memory mapped I/O"



Instructions:

Commands. Seq: "program"

IR – current instr (value)
PC – next instr (address)

1) Fetch next instr READ
2) Decode
3) Fetch operands (if needed) READ
4) Perform
5) Store result (if needed) WRITE

Example:

1) LOAD R2, LOC
   1) FETCH + Increment PC
   2) DECODE IR
   3) MAR ← LOC
   4) READ
   5) R_2 ← MDR

*MIDTERM: How many read/writes for x. instr?
→ Don't forget the READ to fetch instr.

2) STORE R_4, LOC
   1) Fetch, PC++
   2) DEcode IR
   3) MAR ← LOC
   4) MDR ← R4
   5) WRITE

3) C = [A] + [B] . [ ] means read from address. Left side: location.

Instruction Processing:
Mem. read/write = significant bottleneck.

More Registers:
CR: Control Register - control CPU behaviour
SR: Status Register - store flags for status
→ only PC, SR, CR can be directly modified by instructions.
→ NOT IR, MAR, MDR (Hardware does this)

Sep 21, 2018

Risc vs Cisc:

RISC:
- Reduced Instr Set Computers
- Each inst does a simple function
- Simple for compilers
- ARM uses.

CISC: Came before RISC (simple for programmers)
- Complex " "
- complex functions (insert Queen)
- Hard for compilers
- Intel x86 uses.
- Hard to optimize (instrs of diff sizes).

Cisc:

- wastes space, tries to make all instrs same size.
- uses extension words.
  - → How do instrs have diff sizes? can pass 2, 1.5, 1, 0 addresses.
  - → Midterm: know how instrs are executed + special registers.

Memory Addressing:

- "Address space" range of memory addresses. 24 bits? space = 2^{24}.
- "Byte Addressable", addresses are for byte, words = 4 bytes.
  - → Aligned if %4 == 0. word addressable is v. inefficient: size*=4.
- Binary prefixes: $k = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$ ... *ALWAYS load into aligned addrs.
  - → We will use KB, KiB = 1024 bytes. kB = 1000 bytes.

Endianness:

Which byte stored first?
- → Big: Left bytes → low addresses (more sig. bits) www
- → little: Right bytes → low addresses (less sig. bits). ARM
  - → Assume little.

Instr Types:

1) Data Transfers (load/store) mem ↔ processor
2) A/L (add)
3) Seq. and cntrl (branch)
4) I/O (store/load) I/O ↔ processor (mem-mapped)

ISA: Instr. set architecture

Register Transfer Level Notation:

Sept 24, 2018

1) Symbolic names for mem. locations. (LOC1, LOC2)
2) Reg. names for registers (R0, R1)
3) [LOC1] means content @ addr LOC1
4) Control signals: T_1, T_2 (symbolic) (dereference pointer)
5) R1 ← [LOC1] left location, right value!

Example:

1) $R_1 \leftarrow [[R_2]]$ val at addr in R_2 to R_1 can't do $R_1 \leftarrow R_2$
2) $[R_1] \leftarrow [R_2]$ val at R_2 to addr in R_1.
3) $R_1 \leftarrow$ LOC1 R_1 gets LOC1 address.

Assembly Language Notation:

Load, Store, etc.

Immediate Val: Used for constants:

SUB R2, R2, #1 = $R2 \leftarrow [R_2] - 1$

Addressing Modes:

Immediate	#Value	
Reg	R_i	
Abs	LOC	... etc

Immediate Addressing:

Operand is const. #6

Move R0, #200
R0 ← 200

Register Addressing:

Operand is reg. R_2

Add R_2, R_3, R_4
$R_2 \leftarrow [R_3] + [R_4]$

Absolute (direct) Addressing:

operand is in mem. LOC$_i$

Load R_3, LOC
$R_3 \leftarrow$ [LOC]

Register Indirect Addressing:

Operand is stored in reg/mem loc: $[R_i]$ (R_i) } pointers

Load $R_0, (R_i)$: $R_0 \leftarrow [[R_i]]$

2 accesses in risc

Index Addressing:

Sept 26, 2018

Load $R_1, 20(R_0)$: $R_1 \leftarrow [[R_0] + 20]$

→ Data structures (like frame pointers)

Base with Index Addressing:

register holds the offset: Load $R_2, (R_0, R_1)$: $R_2 \leftarrow [[R_0] + [R_1]]$

can also do $R_2, 10(R_0, R_1)$ (adds 10). 3D arrays

PC-Relative Addressing:

Uses PC as the register for index addressing.

Move $R_0, -16(PC)$: $R_0 \leftarrow [PC] + -16$

Auto Increment Addressing:

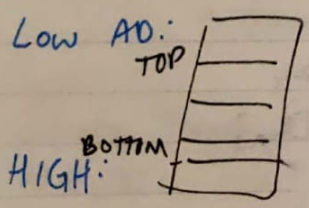
$(R_i)+$ Load $R_2, (R_3)+$: $R_2 \leftarrow [[R_3]], R_3 \leftarrow [R_3] + inc$ (size of R_2)

Post vs pre increment: pre: $+(R_3)$

Auto Decrement:

Self explanatory

Stacks:



STORE $R_2, -(SP)$ push

LOAD $R_2, (SP)+$ pop

Subroutines: Sept 25, 2018

- "call" instruction
  - → Using linkage method. (save return address).
    1) Store contents of PC → link register ($14) (PC post-increment) after call
    2) Branch to target address.
    3) Perform function, then branch back to addr in L.R.
  - → Nesting? Check ahead of time.
    - → SUB1 save LR onto stack before calling SUB2.
    - → Pop when SUB2 done.
    - → Could overflow or be buggy.
- Passing Parameters:
  1) Registers ⟺ low # of params, not recursive, doesn't call another sub w/ same regs
  2) Mem locations Not rlly used.
  3) Stack. if not Reg's, then this
- Must restore used registers in subroutines (that aren't params). Push to stack at beginning of subroutine. Pop at end to restore.
- If using stack for params:
  1) Push params on stack before calling.
  2) In subroutine, push each register used by fcn on stack, then load params from stack (don't pop).
  3) can overwrite params on stack for returned vals.
  4) Restore all registers and restore S. Ptr.
  5) Pop params after getting return value.

Stack Frames:

Frame pointer allows nested subroutines. (register, $11) SP-$12.

- → stack frame: all items on stack prior to calling subroutine and during its execution.
- → Most store frame pointer value on stack when calling.

order ↓

1) Return address (LR) [0) params]
2) FP (copy SP → FP) ← * OLD FP location value, CUR SP Location on stack.
3) Local Vars
4) Reg values to be restored later

-To restore FP, make it point to saved val.
FP = *(FP)

-Returning:

Oct 1, 2018

1) Pop saved reg. vals back
2) De-allocate local vars
3) Save old FP back to FP
4) Restore LR
5) Return

More Instructions:

Logic: and, or, not. 0-extended

Shift/Rotate:

LShift L: logical shift left. → arith. shift preserves sign
Rotate L: MS bits to LS bits.
→ "with carry" uses carry in rotate (1 extra bit)

Assembler:

Converts to m lang.

ARM:

Risc aspects:
- Fixed len instrs
- only LD/STR accesses mem
- ALU instr only on registers

All ARM

Cisc Aspects:
- Auto inc/dec, PC-relative
- condition codes (N, Z, C, etc)
- Multiple reg can be loaded from block of consec mem words, or stored in a block using single instr

ARM Memory:
Byte addr able, words/half/bytes (aligned)
↳ Can only read if % len

Registers:
16 32 bit reg's.
- R15 PC, R13 SP, R14 LR, R11 FP
↳ Just convention
- CPSR/PSR status reg:
↳ N Z C V
↳ N - 1 if res < 0, else 0
↳ Z - 1 if res = 0, else 0
↳ V - 1 if overflow, else 0
↳ C - 1 if carry out, else 0

Addressing Modes: Oct 03, 2018
[Rn, #offset] PC Relative 10(R_2) in RISC
12 bit or another reg.

Pre-Index:
LDR Rd, [Rn, #offset] : $R_d \leftarrow [[R_N] + offset]$
12 bit or another reg

Pre-Index w/ WB:
LDR Rd, [Rn, #offset]! STR R0, [R13, #-4]!
↳ Can also shift in-cmd. R13 becomes R13 - 4, R0 stored.
↳ LDR, R0, [R1, -R2, LSL #4]!
: $R0 \leftarrow [[R_1] - 16[R_2]]$
: $R_1 \leftarrow [R_1] - 16 * [R_2]$

Post-Index:
LDR Rd, [RN], #offset
$R_d \leftarrow [[R_N]]$
$R_N \leftarrow [R_N] + offset$

Mem Operations:

LDR/STR - words
" H - half
" B - bytes
" SH/SB - sign-extended loads (no write)

LDM/SDM - multiple words
→ LDM R10!, {R0, R1, R6, R7}
→ If [R10] = 1000, words at 1000, 1004, 1008, 1012 are loaded into the registers, R10 is 1016 after all transfers.

Arith. Instructions:

ADD R0, R2, R4, LSL #4 — takes 2nd one. $R_0 \leftarrow [R_2] + 4 * [R_4]$

MLA R0, R4, R5, R6 $R_0 \leftarrow ([R_4] * [R_5]) + [R_6]$
can't use offset for this.

Test/Compare Instructions:

TST, TEQ for bitmasks.
TST R3, #1 LSL #31 gets most sig. bit.
→ AND of R3 and 1000...00. sets Z. } sets status bits by default.
TEQ: XOR - equality
→ Doesn't store results! only status bits.
→ SUBS keeps result + cond. code.

Branching:

PC is PC+8 when BEQ executes. If label is used, this is done for you. BEQ: Z=1.
EQ is a suffix. others: NE, MI, PL, etc
- AL is default suffix (BAL). B same as BAL, TST → TSTAL

Comparisons: Oct 5, 2018

CMP A, B ⇔ A − B

UNSIGNED: $A \geq B \Leftrightarrow C=1$ $\quad A > B \Leftrightarrow C=1$ OR $Z=0 \Leftrightarrow (\bar{C}$ OR $Z)=0$

$A < B \Leftrightarrow C=0$ $\quad A \leq B \Leftrightarrow (\bar{C}$ or $Z)=1$

SIGNED: $A \geq B \Leftrightarrow (N=0$ AND $V=0)$ OR $(N=1$ AND $V=1) \Leftrightarrow N \oplus V = 0$

Assembler Directives:

AREA, ENTRY, etc...

Pseudo-Instrs:

= checks if sufficient bits. MOV if enough, else LOAD address.

Push/Pop from Stack: Oct 12, 2018

op {addy-mode}{cond} Rn{!}, reglist

op - LDM/STM

addy-mode - IA: inc after access, DB: dec before access.

Cond - cond exec

Rn: base mem addy

!: optional (final address → R_N)

reglist - low # reg in lowest addy.

LDMFD Pop (LDMIA)

STMFD Push (STMDB)

Multi-pass Assemblers and Linkers:

Refer to CS241e notes.

I/O Device Interface: Oct 15, 2018

Allows CPU to talk to I/O device

- → DATA, CONTROL, STATUS registers for each I/O device. - 1 byte each, but word aligned
- → Connected to Interconnection network
- → Program-controlled } How to know there is data?
- → Interrupt-driven: }