CS241

ANDREW CODISPOTI

1. Binary and Hexadecimal numbers

(1) bit – binary digits 1 and 0 (all computer understands)
(2) byte – 8 bits
(3) word
    (a) machine specific grouping of bits
    (b) assume 32-bit architecture
    (c) 1 word = 32 bits= 4 bytes
(4) nibble – 4 bits half a byte

1.1. **Given a byte(or word) in memory what does it mean?** Could mean many things.

(1) A number (which number?)

1.2. **How can we represent negative numbers?** Simply use a sign bit with 0 for + and 1 for - (Sign-Magnitude representation) but then you have two -1's and arithmetic is tricky

1.2.1. *Two's Complement notation.* Interpret the n-bit number as a an unsigned int. If first bit is 0 done else subtract 2^n

n bits- represent $-2^{n-1} \ldots 2^{n-1}$ with left bit still giving sign. arithmetic is clean, just mod 2^n

We cant tell if a number is signed unsigned or two's complement and we have to remember.

We don't even know if what it means:a number, a character, An instruction (or part of one), Garbage

1.3. **Hexadecimal notation.**

(1) base 16 0-9, A-F
(2) more compact than binary
(3) each hex digit = 4 bits (1 nibble)
(4) e.g. 1100 1001 = C9
(5) NOTATION: 0xC9

1.4. **Mapping from binary to characters.**

1.4.1. *ASCII.* Uses 7 bits

IBM implemented extended ascii to use all 8-bits, but they add some weird characters i.e. frame like characters.Compatibility issues because no one standard.

11001001 is not 7 bit ascii, 01001001 decimal 73 is ASCII for I

other standards like EBCDIC

2. Machine Language

Computer programs operate on data and are data(occupy same space as data)

2.1. **Von Neumann architecture.** Programs reside in the same memory as data. Programs can operate on other programs i.e OS

2.2. **Central Processing Unit.** see physical notes for diagram

(1) Control Unit
   (a) decodes instructions
   (b) dispatches to other parts of the computer to carry out instructions
(2) Arithmetic logic unit: Does Math

2.3. **Memory–Many Kinds (Ranked in speed order).**

(1) **CPU
(2) cache
(3) **main memory RAM
(4) disk memory
(5) network memory

2.4. **Registers.** On the CPU, small amount of very fast memory called registers

MIPS 32 General purpose registers $0 to $31

(1) each holds 32 bits
(2) can only operate on data that is in regs.
(3) $0 is always 0
(4) $31 is special and $30
(5) EX: add the contents of two registers and put the result in another register.
(6) 5 bits encode a register $2^5 = 32$
(7) 15 bits to encode registers, 17 bits to encode operation

2.5. **RAM.**

(1) large amount of memory away from cpu
(2) travels between the cpu and ram on the bus
(3) big array of n-bytes, n 10^9
(4) each cell has an address 0,. . . , n-1
(5) each 4-byte block of the form is a word(see diagram 2 in notes)
(6) word addresses are 0,4,8,c,10,14,18,1c
(7) RAM access much slower than reg access

2.6. **Communicating with RAM.** two commands

(1) load
   (a) transfer a word from an address to a register. desired address goes into the memory address register(MAR), goes out on bus
   (b) data at that location comes back on the bus, goes into the memory data register (MDR)
   (c) value in MDR moved to destination register
(2) store: does the reverse of load

2.7. **How does a computer know which words contain instructions and which contain data?** It Doesn't

2.8. **How does it run.** Special register called pc(program counter) which stores the address of the next instruction to execute instruction to execute.

By convention, guarantee that some address(i.e. 0) contains code, initialize pc to 0.

Computer then runs the fetch-execute cycle

```
PC <- 0
loop
IR <- MEM[PC]
PC <- 4
decode and execute the instruction in IR
end loop
```

only program the machine really runs

NOTE: PC holds the address of the next instruction while the current instruction is executing.

2.9. **How does a program get executed.** Program called a loader that puts the program in memory and sets PC to teh address of the first instruction in the program and sets PC to teh address of the first instruction in the program

2.10. **What happens when a program ends?** need to return control to the loader, set pc to the address of the next instruction in the loader.$31 will contain the right address.

need to set pc to $31

2.11. **Example.** Example1: Add value in $5 to the value in the $7 sotre result in $3 and return

location	binary	hex	meaning
00000000	0000 0000 1010 0111 0001 1000 0010 0000	00a71820	add $3, $5, $7
00000004	0000 0011 1110 0000 0000 0000 0000 1000	03e00008 * jr $31	

Example 2 add 42 to 52, store in $3, return

lis $d "load immediate and skip", treat the next word as an immediate value and load it in to $d then skip to the instruction

location	binary	hex	meaning
00000000	0000 0000 0000 0000 0010 1000 0001 0100	00002814	lis $5
00000004	0000 0000 0000 0000 0000 0000 0010 1010	00000004	.word 42
00000008	0000 0000 0000 0011 1000 1000 0001 0100	00002814	lis $7
0000000c	0000 0000 0000 0000 0000 0000 0011 0100	00000004	.word 52
00000004	0000 0011 1110 0000 0000 0000 0000 1000	03e00008	jr $31

```
add \$3 \$5 \$7
```

2.12. **assembly language.** replace tedious binary/hex encodings with easier to read mnuemonics less chance of error, translation to binary can be automated (assembler), one line of assembly = one machine instruction (word)

```
lis $5; load imm and skip
.word 42; not an instruction, is a directive that next word in binary should
literally be 42
list $7;
.word 52
add $3 $5 $7;destination reg first
jr $31
```

2.12.1. *EX3.* Compute the absolute value of $1 store in $1 and return

(1) some insturcitons modify PC = "branches" and "jumps", i.e. jr
(2) beq: branch if 2 registers have equal contents, increment PC by a given number of words, can branch backwards.
(3) lone bne
(4) slt: "set less than"

```
slt $a, $b, $c
$a = {1 : $b<$c>, 0}
```

2.12.2. *RAM.* lw= load word from ram into reg

```
lw $a, i($b) loads the word at MEM[$b + i]into $a
```

sw = store word from regs into RAM

```
sw $a , i($b) stores word in $a at mem[$b+1]
```

see examples on paper.

2.13. **Multiplication.** mult = multiply

```
mult $a, $b ; Product of 2 32 bit numbers up to 64bitz, too big for register
```

Concatenation of hi and lo registers is the entire product of multiplication
mflo = move from lo

```
mflo $a, ; $a <- lo
```

for division lo stores quotient, and hi stores remainder

2.14. **revisit looping example.** have to keep track of offsets if instuctions are added or remove

2.15. **Assembler.** assembler allows labeled instrcutions

```
foo: add $1, $2, $3
```

assembler associates the name foo with the address of the instruction

assembler calculates the distance between the label and the program counter, in words $\frac{top-PC}{4}$

3. Procedures in MIPS

2 problems to solve

(1) call and return : transferring control into and out of the procedure (Procedures calling other Procedures)
(2) Registers: what if a proc overwrites my registers.

We could : reserve some registers for f and some for main line so they wont interfere. but when using recursion we run out of registers.

Instead guarantee that procs leave regs unchanged when done by storing in RAM. Must stop processes from using the same ram becuase the same issue will arise.

see diagram

Can allocate from one end of ram to the other for procs. Need to track what ram is in use. Mips machine helps us $30 initialized by the loader to just past the last word of memory.

can use $30 as a "bookmark" to separate used and unused RAM

diagram

RAM uses LIFO order $30 is the stack pointer address at the top of the stack

3.0.1. *Template for Procedures.*

```
f:  sw $2, -4($30) ;
    sw $3, -8($30); push registers f modifies on the stack
    lis $3
    .word 8   ; decrement 30
    sub $30, $30, $3
    ; body

g:
    add $30, $30, $3 ; assuming $3 remains 8 increment 30
    lw $3, -8($30)
    lw $2, -4($30)
```

3.0.2. *Call and return.*

```
main: ...
lis $5
.word f; address of line labelled f
jr $5 ; jump to that line
```