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 $0, \ldots,$ 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

3.1. **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)
```

3.2. Call and return.

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

Return: we need to set PC to the line after the jr(i.e. to HERE)
Solution: jalr

3.2.1. *jalr(Jump and link regiter).*

like jr, but sets $31 to the address of the next instruction

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

Question: jalr overwrites $31 so how do we get back to the loader, what if f calls g
Answer: Save $31 to the stack before the call and restore afterwards

```
main:
lis $5
.word f
sw $31, -4($30)
lis $31
.word 4
sub $30, $30, $31
jalr $5
lis $31
.word 4
add $30, $30, $31
lw $31, -4($30)
jr $31
```

3.3. Parameters and Results.

generally use regs (document)
if too many, use stack

```
; sum 1ToN: computes 1 + ... + N
; Register
; $1 - working
; $2 - input (value of N)
$ $3 - output; do not save this one
sum1toN:
```

4. Recursion

no extra machinery needed
if registers, parameters, stack managed properly, recursion will just work

4.1. **I/O.** output: Use sw to store word in location 0xffff000c. the last byte in the word will be printed

```
lis $1
.word 0xffff000c
list $2
.word 67
sw $2, 0($1)
lis $2
.word 83
sw $2, 0($1)
```

5. The Assembler

Any translation involves two phases

(1) Analysis: Understand what is meant by the source string
(2) Synthesis: output equivalent target string

starts with assembly file: stream of characters

(1) group characters into meaningful tokens: label, hex #, reg #, .word, etc
(2) group tokens into instructions if posssible
(3) if tokens do not form sensible instructions, output ERROR to stderr
(4) NOTE: There are many more wrong tokens than right ones. try to find right combos

5.1. **Biggest problem with assembler.** how do we assemble

```
beq $2, $0, abc
abc:
```

cant assemble the first beq because we dont know the value of abc

5.2. **Standard solution.** assemble in two passes

Pass 1: group tokens into instructions. record addresses of labelled instructions into a symbol table(list of [label, address] pairs)

NOTE: a line of assembly can have more than one label. you can label the word after the end of the program

Pass 2: translate each instruction into machine code. If an insturction refers to a label, lookup the associated address in the symbol table

Your Assembler output the assembled mips code to stdout. output the symbol table to stderr

marmoset only cares outupt ERROr

5.3. **Code.**

```
main: lis $2
.word 13
add $3, $3, $0
top: add $3, $3, $2
lis $1
```

```
.word 1
sub $2, $2, $1
bne $2, $0, top
jr $31
end:
```

5.3.1. *Pass 1.* group tokens into instructions. build symbol table

main	0
top	c
end	24

5.3.2. *Pass 2.* translate each instruction

```
lis $2  ---- 0x00001014
.word 13 ---- 0x0000000d
bne $2, $0, top -- lookup top in symbol table
calculate (top-pc)/4 = -5 ==> 0x1440fffb
```

6. Bit level operations

to assemble bne \$2, \$0, top (where (top - pc)/4 = -5)

```
opcode = 000101 = 5
1st reg  = $2 = 00010
2nd reg = $0 = 00000
offset = -5
|6 bits (opcode) | 5 bits(1st reg) | 5 bits(2nd reg) | 16 bits (offset)
```

to put 000101 into the first 6 bits append 26 0's (left shift by 26 bits) 5 « 26

move 2 21 bits to the left
2 « 21
move 0 16 bits to the left
0 « 16

6.1. **Bitwise and/or.** normal AND/OR.

and with 0 gives 0, with 1 gives other digit
or with 0 gives other digit, 1 gives 1
use bitwise and to turn bits off
use bitwise or to turn bits on
bitwise and with 0xffff, -5 & 0xffff bitwise and -5
bitwise or the four pieces together
(5 « 26) | (2 « 21) | (0 « 16) | (-5 & 0xffff)

6.2. **C++ stuff.** int converts to int ascii codeo

char outputs the actual value and the screen interprets

```
unsigned int instr = 5 << 26 |...;
char c = instr >> 24;
cout << c
```

```
c = instr >> 16;
cout << c;
c = instr >> 8;
cout << c;
```

7. Loaders

OS Code:

```
repeat:
p <- next program to run
copy P into memory, starting at 0
jalr $0
beq $0, $0, repeat
```

8. OS

Problem: os is a program - where does it sit in memory. other programs in memory at the same time, all cant be at address 0.

labels may be resolved to the wrong addresses

8.1. How do you fix it.

could pick different starting addresses for programs at assembly time

let the loader decide where to put the program, fix bad label references

8.2. Loader's job.

(1) take a program P as input, find a location a in memory for P
(2) copy P into memory starting at a
(3) return a to the os

```
OS 2.0
repeat: p <- next program
$3 <- loader(P)
jalr $3
beq $0, $0, repeat
```

9. Loader Pseudocode

Input: words $w_1, \ldots, w_n <$ - the code
n = k + space for stack (how much? pick something)
a = address of n contiguous words if free RAM

```
for i-0..k-1
MEM[a+i*4] <- wi+1
$30 a+4*n
return a
```

9.1. **What needs to change when we relocate?**

(1) offset added in order to fix the askew entries. add alpha to word
(2) dont adjust constant word values
(3) do not adjust everythign else including branches

9.1.1. *Problem.* Assmbled file is a stream of bits, how do you know it came from .word(with an id!) and which are instructions

We cant do this, and we need more info from the assembler

output of most assemblers are not just machine code, it also produces object code Object file: contains binary code AND auxiliary info needed by the loader and linker

we have our own object code format called MERL(Mips executable relocatable linkable)

9.1.2. *What do we need to put in our object file.*

(1) the code
(2) which lines of code(addresses) were originally .word id

10. MERL format

start at 12 because header starts at 0 and it consists of 3 words

10.1. **Want assembler to generate relocatable object code.**
Relocation Tool: cs241.merl ==> takes in MERL file and relocation address, outputs non relocatable mips file with header and footer removed ready to load at the given address

mips.twoints/array: optional second argument = address at which to load the file

E.g.: load myobj.merl at 0x10000

```
java cs241.merl 0x10000 < myobj.merl > myobj.mips
java mips.twoints myobj.mips -x10000
```

loader relocation algorithm

```
read()//skip cookie
endMod <- read() -12 //length of code and footer
codeLen <- read() -12 //length of code
alpha <- findfreeRAM(codeLen+stack)
for(i=0; i<codeLen;i+=4>)
  MEM[a+i] = read()
end for
while(i<endMod)
format <- read()
  if(format == 1)
    rel <- read()/address to be relocated
    MEM[rel+alpha - 12]/*actual location in RAM (header not
    loaded)*/+= alpha - 12    //adjust forward by alpha backward by header
  length 12
  else ERROR
  i+=8
```

```
end while
```

11. Linkers

Issue: how can the assembler resolve a reference to the label in a different file
soln1: cat the files, assemble the result
soln 2: tool that understands Merl files and puts them together intelligently - a linker
what should the assembler do with references to labels that aren't there?

(1) need to change the assembler
(2) when the assembler encounters .word id where label id is not found, it fills in 0 and indicates that the program requires the value of id before it can run

11.1. How does the assembler notify us?

makes an entry in the MERL file
we lose a valuable error check by having to look at different files for label definitions

11.2. How can the assembler knwo what is an error and what is intentional?

Create a new assembler directive:

```
.import id
```

tells the assembler to ask for id to be linked in.
Does NOT assemble to a work in MIPS
When the assemmbler encounters `.word abc` if label abc: is not found and no .import abc, then error. Notifyins us of imported symbols MERL entry
Format code 0x11 means External symbol reference(ESR)

11.3. What information must be recorded?

(1) Where, at what address is the symbol being used? (where is the 0 that we need to fill in)
(2) what is the name of the symbol

Format of an ESR entry:

```
word1 - 0x11
word2 location where the symbol is used
word3 length of the name in characters
word4
.
.       ASCII chars in the symbol's name (each char in its own word)
.
word 3+n
```

11.4. How can the linker know which abc to link to?

labels wil sometimes be duplicated

11.5. How can we make abc in b.asm unavailable.

another assembler directive and MERL entry type
does not assumble to a word of mips, tells te assembler to make an entry in the MERL symbol table

11.5.1. *Entry Type: External Symbol Definition(ESD).*

```
word1 - 0x05 - formate code for ESD
word2 - address the symbol represents
word3 - length of the name in chars(n)
word4 -

.     The name in ASCII, one word per char
.
word3+n
```

12. Linker Algorithm

Input: merl files m1 and m2
Output: single merl file with m2 linked after m1

```
a <- m1.codelen-12
relocate m2.code by \alpha
add \alpha to every address in m2.symtbl
if m1.exports.labels \cap m2.exports.labels != empty ==> ERROR
foreach <addr1,label> in m1.imports
  if(\exists<addr2, label> in m2.exports)
    m1.code[addr1] <- addr2
    remove<addr1, label> from m1.imports
    add addr1 to m1.relocates
foreach <addr2,label> in m1.imports
  if(\exists<addr2, label> in m2.exports)
    m1.code[addr2] <- addr1
    remove<addr2, label> from m2.imports
    add addr2 to m2.relocates

imports = m1.imports \cup m2.imports
exports = m1.exports \cup m2.exports
relocates = m1.relocates \cup m2.relocates

Output MERL cookie
output total code length + total symbol table length +12
output m1.code
output m2.code
output imports, epxorts, relocates
```

13. Formal Languages

High Level Lang –> Compiler –> assembly

Assembly: simple structure. easy to recognize and parse. straighforward unambiguous translation to machine language

High level language: more complex structure, harder to recognize, usually no single translation to machine language

To handle the complexity - a formal theory of string recognition - general principles applicable to programming language

13.1. **Definitions.**

(1) alphabet: finite set of symbols(e.g a,b,c) : denoted σ as in $\sum a, b, c$
(2) string(or word): finite sequence of symbols (from σ)e.g. a, aba, cbca, abc
(3) length of a word |w| : number of characters in the word e.g. |aba|=3
(4) empty word: an empty sequence of symbols ϵ length of epsilon is 0, epsilon denotes empty string
(5) language: set of strings (words); i.e. $a^{2n}b, n \geq 0 ==>$ words consisting of an even number of a's followed by b
(6) NOTE: ϵ - empty word or $\emptyset$- empty language; ϵ - singleton language that contains only 2

13.2. **How can we recognize automatically whether a given string belongs to a given language?** Depends on how complex the language is. $a^{2n}b, n \geq 0$ is easy, valid mips assembly programs easy, valid c programs, harder, some languages impossible

Characteristize languages according to how hard the recognition process is.

Organize languages into classes of languages based on how hard they are to recognize

(1) finite
(2) regular
(3) context-free
(4) context-sensitive
(5) recursive
(6) etc.

languages get harder to recognize as you go down the list. start from easy to impossible

have only finitely many words
can recognize word by comparing with each word in the finite set
can we do this more efficiently?

13.3.1. *Exercise.* L = cat, car, cow.Write code to answer $w \in L$ such that w is scanned exactly once without storing previously seen characters.

```
scan input left to right
if first char is c, move on, else error
if next char is a
__if next char is t
____if no more chars, accept else error
__else if nex char is r
____if no more chars, accept else error
```

```
__else error
else if next char is o
__if next char is w
____if no more chars accept else error
__else error
else error
```

An abstraction of this program

can generalize into state machine: bubbles are called states, configurations of the program based on input seen

not very useful to use this because languages usually dont have finitely many programs

13.4. **Regular Languages.** build from:

(1) finite languages
(2) union,$L_1 \cup L_2 = x|x \exists L_1, or x \exists L_2$
(3) concatenation $L_1 \cdot L_2 = x, y|x \exists L_1, y \exists L_2$
(4) repetition L* $= \epsilon \cup x \cdot y|x \exists L*, y \exists L = \epsilon \cup L \cup LL \cup LLL$ 0 or more occurences of words in L