Topic 2 – MIPS Assembly Language

Key Ideas

- High Level Language vs. Assembly Language vs. Machine Code
- opcodes (operation codes) and operands
- CS241 subset of the MIPS32 instruction set

References

- CO&D Chapter 2 *Instructions: Language of the Computer*

Overview

High Level Language - HLL

- e.g. C, C++, Racket, Python

Assembly Language - AL

- e.g. MIPS, x86-64, ARMv7

Machine Code - MC

- sequence of 0's and 1's associated with a particular processor

b = 10 + a;

↓

lis $1
.word 10
add $3, $2, $1

↓

0000 0000 0000 0000
0000 1000 0001 0100
0000 0000 0000 0000
0000 0000 0000 1010
0000 0000 0100 0001...

Overview

High Level Language (HLL)

- meant to be read and *understood by humans* (smart ones anyways ;-)
- meant to be as *convenient as possible for computer programmers*
- processor independent
  - e.g. can use C++ for many difference processors
- a single statement in a HLL may be translated to several statements in Assembly Language
- most programmers program in a HLL

Overview

Machine Code (MC)

- meant to be *executed by processors*
- meant to be as *convenient as possible for computer hardware*, e.g. binary encoding, 2's complement
- processor dependent: machine code that works for an Intel Core i7 won't work on an ARM processor
- no sane person (except as a learning experience) programs in machine code
- also called Machine Language

Overview

Assembly Language (AL)

- meant to be a *compromise between a HLL and MC*
- it is MC with simple modifications so that humans can understand it easier (e.g. written in mnemonics , assembler directives, labels).
- for the most part, a single statement in AL is translated to a single statement in machine code
- you can take AL for one processor and run it on another (that's what we'll be doing) using a simulator
- only a small minority of programmers program in AL

MIPS Architecture

What is MIPS

- MIPS is one particular family of processors
- popular, simple and easiest to study
- multiple revisions exist, e.g. MIPS I, MIPS II, MIPS III, ...
- it has evolved over time $\Rightarrow$ not just a single standard
- the version we will be looking at, MIPS32, is a 32-bit architecture

MIPS32 Assembly Language

Word Size

- 32-bit architecture means its *word* size is 32 bits
- pathways from one component to the next transfer 32 bits in parallel
- this size is typical of processors in smart phones and tablets
- 64-bit architecture is typical in laptops, desktops and servers
- for MIPS AL, each instruction takes exactly 32 bits
  - other processors can have variable length instructions, i.e. some longer than 32 bits

C++ vs. MIPS Assembly Language

C++ code:

```
a = 10;
b = 15;
c = a + b;
```

```
lis $5              ; load the next word into register 5
.word 0xa           ; a is hexadecimal for 10
lis $7              ; load the next word into register 7
.word 0xf           ; f is hexadecimal for 15
add $3, $5, $7      ; register 3 = register 5 + register 7
jr $31              ; jump to the address stored in $31
```

High Level vs. Assembly Language

Assembly Language

- one statement per line
- uses mnemonics for statements, e.g. *lis* for load immediate and skip, *jr* for jump (to address stored in) register
- *big difference: AL uses registers rather than variables to hold data temporarily and manipulate it* (e.g. *$3*, *$5*, *$7*)
- can have a huge number of variables in a HLL (no practical limit really) but there are only a limited number of general purpose registers in AL
- each register holds 32 bits
- for MIPS there are 32 registers, called $0 .. $31
- a typical value for the number of registers in many current processors is around 16 (e.g. ARMv7 and x86-64)

High Level vs. Assembly Language

Arithmetic Operators and Registers

- In a *High Level Language* you typically manipulate data in terms of variables and arithmetic operators

  total = subtotal + GST;

  root1 = (-b + sqrt((b**2) – (4*a*c))) / (2*a);

- In *Assembly Language*
  - use words (mnemonics): *add*, *sub*, *mult*, *div* rather than symbols +, -, *, /
  - specify registers, e.g. $2, rather than variables
  - some registers have a specific purpose
  - in MIPS, we reserve $30 for stack pointer (SP) and $31 for a return address, and $0 always contains the value zero

Machine Code

What is Machine Code (MC)

- binary code – comprised of 0s and 1s
- directly executed by the processor
- the program (a sequence of bits) is split into instructions with the following format:
  - operation code (*opcode*) + *operands*
  - instructions specify what operations the processor should execute and where the data is
    - *opcode* designates the *operation*, say add or sub
    - *operands* designate the *data sources and destinations*, which are either registers or memory locations (RAM)
- e.g. in AL add $d, $s, $t means set the value in $d to be equal to the value in $s plus the value in $t (i.e. $d = $s + $t)

Machine Code

Example: add

in AL: add $d, $s, $t

in MC: 000 00ss ssst tttt dddd d000 0010 0000

- opcode
  - in AL: add
  - in MC: 000000 ____ ____ ____ 0000010 0000
- operands
  - in MC: *sssss*, *ttttt*, and *ddddd* are binary numbers between 0 and 31 that specify which registers ($0 to $31) store the data for the add operation and where to place the result
  - $2^5 = 32$, so it takes 5 bits to specify the 32 registers
  - typically *s* and *t* are called the *source registers* and *d* is called the *destination register*

Machine Code

Example: add vs. sub

- add $d, $s, $t in AL is the following in MC
  0000 00ss ssst tttt dddd d000 0010 0000 and
- sub $d, $s, $t in AL is the following in MC
  0000 00ss ssst tttt dddd d000 0010 0010
- the *opcode* is a pattern that turns on and off various components of the processor so that whatever flows to the Arithmetic Logic Unit (ALU) will be added (if 2^{nd} last bit is not set) or subtracted (if 2^{nd} last bit is set)
- the operands $s and $t signal which register values should flow into the ALU to be added or subtracted

Instruction Set

Varieties of Instruction Sets

- an *instruction set* is the repertoire of *instructions understood by a processor*
  - e.g. *add*, *sub*, *lis* (load immediate and skip) and *jr* (jump register) that we saw in the samples of MIPS assembly language
- different processors have different instruction sets but they would have many commonalities

Some Basic MIPS AL Instructions

Addition and Subtraction

```
add $3, $1, $2
```

- i.e. \$3 = \$2 + \$1
- add (the contents of) register \$1 and \$2
- place result in register \$3
- often use the notation: add \$d, \$s, \$t where
  - \$s and \$t are the source
  - \$d is the destination

```
sub $d, $s, $t
```

- i.e. \$d = \$s - \$t
- subtract (the contents of) register \$t from (the contents of) \$s
- place result in register \$d

Some Basic MIPS AL Instructions

Arithmetic Operations, e.g. add

- have two sources (of data) and one destination (for the result)

  C / C++: r1 = r2+ r3;

  MIPS : add $1, $2, $3

- the destination can be the same as one of the sources

  C / C++: r1 += r2;

  C / C++: r1 = r1 + r2;

  MIPS : add $1, $1, $2

Some Basic MIPS AL Instructions

Arithmetic Operations, e.g. add

- complex expressions must be broken up into simpler expressions with two source operands and one destination

C / C++: r1 = r2+ r3 + r4 + r5

means r1 = (((r2+ r3) + r4) + r5)

MIPS :

```
add $1, $2, $3
add $1, $1, $4
add $1, $1, $5
```

Some Basic MIPS AL Instructions

Constants

- to load in a constant `i` use the `lis` and `.word` combination

```
lis $d
.word i
```

- `lis` means *load immediate and skip*
  - load the next value (in this case `i`) into $d and then skip (i.e. don't try and execute) the next word
  - interpret `i` as data rather than as an instruction
  - not an actual MIPS instruction, but a *pseudo instruction*, i.e. it is provided as a convenience and gets converted into other MIPS instructions (a variant of this is called `li`)
- `.word` means store the value `i` right after the `lis $d` instruction

Some Basic MIPS AL Instructions

Jumping

```
jr $s
```

- meaning: jump (to the address in) register $s
- start executing code at this new location
- *used to implement returning from a function call*
  - load my current address into $s
  - then call the function, i.e. go to a different address
  - when the function is done, I need to return to the address (or location) when I came from so I execute `jr $s`
- E.g. there any many places in C++ code where I would call decltype. Each time I call it, I first need a store my current location so when decltype is done, it knows where to return to.
- Convention: for a function, register $31 holds the address you return to after the function is done