Lecture 2 Notes
Connor Baker, January 19th 2017

Agenda

The agenda for today's lecture is as follows:

1. Arithmetic operations
2. Logical operations
3. Data transfer

1 Programming Hierarchy

To convert a high level language (HLL) into its machine readable equivalent, the compiler: (see Figure (1))

1. Converts the HLL into *assembly language* (ASM)
2. Translates the ASM into 0's and 1's for the hardware
3. Writes the translated form to an *object module* (a file with the extension .obj)
4. A *linker* converts the object module into an executable file, formatted for the target operating system
   (a) The linker also grabs libraries used by the program to be included in the executable
5. The executable file is loaded by the OS and executed directly by the hardware

1.1 What is ASM?

1. The symbolic representation of 0's and 1's the CPU understands
2. ASM allows humans to easily read and interpret CPU instructions
3. ASM is considered a low level language (LLL)
4. ASM is not compiled – it is assembled by the assembler

1.2 HLL vs ASM

HLL's such as Java, C, C++ are all machine independent. LLL's like ASM are all machine dependent. The language of the ASM output is determined by the target hardware. See Figure (2) for how Java runs everywhere

1.3 ASM - General

1. HLL's have verbs, statements, declarations, etc.
2. LLL's have *instructions* and *operands*

   Instructions: The "words" of a machine language

   Instruction Set: The entire list of instructions recognizable by the computer

1.4 The MIPS CPU

1. A reduced instruction set computer (RISC)
2. The basis for your learning in this class
3. MIPS CPUs can be found as embedded processors in game consoles and TVs
4. We will learn MIPS ASM
   (a) We must learn not only the operand set, but also the register set

1.5 MIPS ASM

1. General format:
   (a) {lable}: instruction operand{operand,{operand,{...}}} # comment
2. Fields within braces are optional
3. Field descriptions

Label	names a particular instruction
Instruction	symbol for the machine instruction
Operand	register, value, or label

1.6 MIPS Registers

1. Registers are temporary storage locations within the CPU
2. Used by CPU hardware for processing instructions
3. Provides the data needed for performing operations
4. Provides a place to receive data from main memory or a source for data to be stored to main memory

5. Every CPU has a set of registers (a register set)
   (a) Most CPUs have a different number of registers

1.7 MIPS Register Set

1. 32 registers total, each 32 bits in size
2. Common registers
   (a) $T0-$T7
   (b) $S0-$S7
3. Special registers
   (a) $Zero,$AT,$GP,$FP,$RA
4. Other registers
   (a) $V0-$V1
   (b) $A0-$A3
   (c) $T8-$T9
   (d) $K0-$K1

1.8 MIPS ASM

1. Consider this HLL code example:

   $$F = (G + H) - (I + J)$$

2. Assume the following:
   (a) F is $S0
   (b) G is $S1
   (c) H is $S2
   (d) I is $S3
   (e) J is $S4
3. The MIPS equivalent code for this statement is:

   add $T0,$S1,$S2 # $\$T0 = G + H$
   add $T1,$S3,$S4 # $\$T1 = I + J$
   sub $S0,$T0,$T1 # $F = (G + H) - (I + J)$

   NEVER ASSUME THAT A REGISTER IS EMPTY.

1.9 Registers, variables, and memory

1. How did the registers become associated with these variables?
   (a) Remember that variables are stored in the main memory
   (b) Somehow we must get data from main memory into a register and vice versa
2. To do this, we must first know where in memory the variable is stored
   (a) This is called the address of the variable
3. We then use data transfer instructions to move data back and forth
   (a) These instructions use the address you provide and the register you specify to move the data
4. MIPS provides two basic data transfer instructions, LOAD WORD and STORE WORD
5. Using the last example $F = (G + H) - (I + J)$

```
 lw  $S1,G
 lw  $S2,H
 lw  $S3,J
 lw  $S4, J
add  $T0,$S1,$S2 $T0 = G + H
add  $T1,$S3,$S4 $T1 = I + J
sub  $S0,$T0,$T1 F = (G + H) - (I + J)
 sw  $S0,F
```

6. LOAD WORD basically functions as a pointer to a memory location

1.10 Arrays

1. Consider $G = H + A[8]$
2. How do we get to the eighth element of array A?
3. We first need to know the base address (where A starts)
   (a) This is the zeroth element ($A[0]$)
   (b) Place this value into a register, let's call it $S3

4. We then use an offset to get to the appropriate element

   lw $TO,32($S3) # $T0 gets $A[8]$

5. Notice that the offset is 32 (since each array item is 4 bytes)
   (a) In MIPS, each array element (which is variable) is 32 bits in size (4 bytes)
   (b) Therefore, the eighth element is the number of bytes times the item desired ($4 \times 8 = 32$ bytes from the beginning of the array)
6. Offsets can be positive or negatives
   (a) This allows us to go forward or backwards through an array
7. See Figure (3)

1.11 MIPS Memory Data Alignment

1. All variables in memory are aligned to addresses divisible by 4 (32 bits)

   $0, 4, 8, 12, 16, 20, 24, 28, 32, \ldots$

2. This is done for speed
3. This is known as an alignment restriction
4. So how is data stored in memory?
   (a) Little-endian (Intel): The least significant byte (LSB) is stored at the lowest address
   (b) Big-endian (MIPS): Most significant byte stored at the lowest address

1.12 Big-endian vs Little-endian

1. Consider this 32 bit value

12345678_{16}

Address	0	1	2	3
Big-endian	12	34	56	78
Little-endian	78	56	34	12

2. This means that numerical dumps of data from our programs will appear to be backwards

1.13 Immediate Values

1. Remember this instruction?

   lw $T0,32($S3) # $T0 gets $A[8]$

2. The value 32 is constant in the instruction
3. Constant values stored in instrutions are known as immediate values
4. They are called such because there is no memory fetch required to obtain them; they are placed directly in the instruction itself

1.14 Immediate Values in MIPS

1. Can be signed or unsigned
2. Limited in size, typicall 16 bits
3. Read-only values
4. Can only be used in certain instructions

1.15 MIPS Instruction Types

1. MIPS have three basic instruction types:

Type	Operand 1	Operand 2	Operand 3
R-type	Register	Register	Register/Immediate*
I-type	Register	Register & Immediate	N/A
J-type	Immediate (address)	N/A	N/A

*Immediate values permitted only within certain instructions

2. The J-type is just an OPcode followed by an address (6 and 26 bits allocated respectively)

1.16 Arithmetic Instructions

1. Add (R-type)

   add $reg1,$reg2,$reg3 # reg1=reg2+reg3

2. Add immediate (I-type)

   addi $reg1,$reg2,imm # reg1=reg2+imm

3. Subtract (R-type)

   sub $reg1,$reg2,$reg3 # reg1=reg2-reg3

   Note 1: Register order is important! Subtraction is not commutative

   Note 2: There is no subi instruction

1.17 Logical Instructions

1. Shifting (R-type)

   sll shift left logical

   srl shift right logical

2. Shifts bits left or right, filling in with zeros

   sll $T2,$S0$4 # shift $S0 left by 4, result in $T2

   srl $T1,$S!$4 # shift $S1 right by 4, result in $T1

3. Uses

   (a) Isolating specific bits

   (b) Multiplying by 2^n (shift left)

   (c) Dividing by 2^n (shift right)

   (d) Bits shifted out of the register are gone

4. Logical AND

   and $reg1,$reg2,$reg3 # reg1=reg2 AND reg3

   andi $reg1,$reg2,imm # reg1=reg2 AND imm

5. This instruciton performs a bitwise AND operation between reg2 and reg3

   (a) Bit 0 of reg2 AND bit 0 of reg3 = bit 0 of reg1

   (b) Bit 1 of reg2 AND bit 1 of reg3 = bit 1 of reg1 ...

   (c) Bit 31 of reg2 AND bit 31 of reg3 = bit 31 of reg1

6. The immediate form uses a 16 bit value (zero extended to 32 bits) with is AND'ed to reg2

7. Uses

   (a) Clearing bits

(b) Isolating bit fields (extracting specific bits)

8. Logical OR

```
or  $reg1,$reg2,$reg3 # reg1=reg2 OR reg3
ori $reg1,$reg2,imm # reg1=reg2 OR imm
```

9. This instruciton performs a bitwise OR operation between reg2 and reg3
   (a) Bit 0 of reg2 OR bit 0 of reg3 = bit 0 of reg1
   (b) Bit 1 of reg2 OR bit 1 of reg3 = bit 1 of reg1 ...
   (c) Bit 31 of reg2 OR bit 31 of reg3 = bit 31 of reg1
10. The immediate form uses a 16 bit value (zero extended to 32 bits) with is OR'ed to reg2
11. Uses
    (a) Setting bits
12. Logical negated OR (NOR)

```
nor $reg1,$reg2,$reg3 # reg1=reg2 NOR reg3
```

13. This instruciton performs a bitwise NOR operation between reg2 and reg3
    (a) Bit 0 of reg2 NOR bit 0 of reg3 = bit 0 of reg1
    (b) Bit 1 of reg2 NOR bit 1 of reg3 = bit 1 of reg1 ...
    (c) Bit 31 of reg2 NOR bit 31 of reg3 = bit 31 of reg1
14. Uses
    (a) Can be used to perform a NOT operation
        i. X NOR 0 (zero) = NOT X

1.18 Data Transfer

1. Load or Store word

```
lw $reg1,imm($reg2)
sw $reg1,imm($reg2)
```

2. Fetches (or stores) the 32 bit memory contents pointed to by the address in reg2, offset by the immediate value
   (a) Target memory address = $reg2+immediate
   (b) $reg1 = contents of target memory address (LW)
   (c) Contents of target memory address = $reg1 (SW)

1.19 Take Away

1. All terms in italics
2. Conversion of HLL code to MIPS ASM
3. MIPS registers (start memorizing the green card!)
4. Variable memory relationship
5. Array element access (specifically how its done in MIPS)
6. Alignment restriction in MIPS
7. Big and little endian
8. Immediate values (purpose and usage)
9. MIPS instruction types
10. All MIPS instructions discussed (how to write them, whay they do)