Recitation 2 – Part A

Assembly (MIPS):

Translation from C to Assembly
(Simple blocks, conditions, loops)

Overview of the course

- Computer Abstractions and Technology
- Instructions: Language of the Computer
- Arithmetic for Computers
- The Processor
- Large and Fast: Exploiting Memory Hierarchy
- Storage and Other I/O Topics

Definitions

- ISA – ISA is an abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program that will run correctly.
- Instruction Set – The vocabulary of commands understood by a given architecture. The words of computers language are called instruction.
  Above this machine level is assembly language, a language that human can read.
- Assembler – The assembler translate the instructions into the binary numbers that machines can understood.
- Clock Rate – Amount of cycles per second.
- Compilation – Compilers-The translation of a program written in a high-level language, such as C,C++,JAVA onto instructions that the hardware can execute.

Basic premise

- C code allows us to preform several operations in one line. The processer on the other hand, **can't** preform several operation in one cycle. This is why assembly code must contain one operation per line.
- Let's look at a c code:

$$f = (g + h) - (i + j);$$

- In order to complete this calculation we first need to put the variables in **memory** the processor can use. Next we need to preform two sets of **addition** and place the results in temporary memory. Then we need to **subtract** the results we got and put the result in the memory that belong to f.

MIPS32 ISA

In MIPS32, each instruction in 32-bit long.

We can classify the instructions into 3 groups:

- R-Format Instructions:

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Example:
add $rd, $rs, $rt

- I-Format Instructions:

opcode	rs	rt	Constant/Address (immediate)
6 bits	5 bits	5 bits	16 bits

Example:
lw $rt, const($rs)

- J-Format Instructions: (will be discussed next week)

opcode	Address
6 bits	26 bits

Basic MIPS instructions

- R-Format Instructions
  - add $reg1, $reg2, $reg3
  - sub $reg1, $reg2, $reg3
  - mul $reg1, $reg2, $reg3
  - and $reg1, $reg2, $reg3
  - or $reg1, $reg2, $reg3
  - nor $reg1, $reg2, $reg3
  - slt $reg1, $reg2, $reg3
  - sll $reg1, $reg2, const
- I-Format Instructions
  - addi $reg1, $reg2, const
  - sw $reg1, const($reg2)
  - lw $reg1, const($reg2)
  - beq $reg1, $reg2, Label
  - bne $reg1, $reg2, Label
  - slti $reg1, $reg2, const

MIPS – Design Principles

- **"Smaller is faster":**
  The size of a register in the MIPS architecture is 32 bits. Groups of 32 bits occur so frequently that they are given the name word in the MIPS architecture.
- A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther…
- In MIPS we have 32 registers (of 32 bits each).

MIPS Registers

Name	Register Number	USE
$zero / $0	0	The constant value 0
$at	1	Assembler Temporary
$v0 - $v1	2 - 3	Values for Function Results
$a0 - $a3	4 – 7	Arguments for Functions
$t0 - $t7	8 – 15	Temporaries
$s0 - $s7	16 – 23	Saved Temporaries
$t8 - $t9	24 – 25	Temporaries
$k0 - $k1	26 - 27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Data transfer instructions

- To access a word in memory, the instruction must supply the memory address.
- In MIPS, the memory is byte-addressable (meaning: each address in the memory holds a single byte of data).
- Also, words must start at address that are multiple of 4. This requirement is called an alignment restriction.

- Example: Assume A is an array of **integers**, its address is saved in register $s3. How can we access A[8]?
  - Option 1: lw $t0, 32($s3)
  - Option 2: addi $t0, $s3, 32
    lw $t0, 0($t0)
  - **Note:** The constant in a data transfer instruction (32) is called the offset, and the register added to form the address ($s3) is called the base register.

Question 1 – Arrays & Offsets

Consider the following C code: (Assume b and c are arrays of integers):

```
a = b[10] - c[5];
b[7] = a - c[3];
```

Where the addresses of b[0] and c[0] are stored in $s0, $s1 correspondingly and the value of a is in $s2. Translate the code to assembly.

Question 1 – Solution

```
a = b[10] - c[5];
b[7] = a - c[3];
```

The addresses of b[0] and c[0] are stored in $s0, $s1 correspondingly and the value of a is in $s2.

MIPS Code:

Instruction	**Comment**
lw $t0, 40($s0)	# load b[10] to $t0
lw $t1, 20($s1)	# load c[5] to $t1
sub $s2, $t0, $t1	# put b[10]-c[5] in $s2
lw $t0, 12($s1)	# load c[3] to $t0
sub $t0, $s2, $t0	# put a-c[3] in $t0
sw $t0, 28($s0)	# store the result in b[7]

Question 2 – C to MIPS

Consider the following C code (Assume a, b, c and d are integers):

```
a = 4*d + 2*(b+c);
```

1. Translate the code to MIPS assembly given that a, b, c and d are stored in $s0-$s3 (Without using MUL instructions).
2. What problems could arise from this implementation?
3. The registers $s0-$s3 now contain the <u>addresses</u> of a, b, c and d. Translate the code to MIPS assembly.

Question 2 – Solution

```
a = 4*d + 2*(b+c);
```

1. Translate the code to MIPS assembly given that a, b, c and d are stored in \$s0-\$s3 (Without using MUL instructions).

MIPS Code:

`add $t0, $s1, $s2`	`# put b+c in $to`
`sll $t0, $t0, 1`	`# Multiply b+c by 2`
`sll $t1, $s3, 2`	`# put 4*d in $t1`
`add $s0, $t1, $t0`	`# put 4*d + 2*(b+c) back in $s0 (The new value of a)`

- Each line of assembly language can contain at most one instruction.
- It's the compiler's job to associate program variables with registers.
  - During the course (HW, exam) **YOU** will be the compiler and you will be responsible to choose the registers correctly.

Question 2 – Solution

2. What problems could arise from this implementation?

If d or b+c are too large, using sll could cause a mistake!

For example:

If b + c =

0100 0001 1011 0011 0010 0000 1001 0001 ($1,102,258,321_{10}$)

by shifting it left we will get

1000 0011 0110 0110 0100 0001 0010 0010

which is negative number!

Question 2 – Solution

3. The registers $s0-$s3 now contain the addresses of a, b, c and d. Translate the code to MIPS assembly.

MIPS Code:

lw $t1, 0($s1)	# load b to $t1
lw $t2, 0($s2)	# load c to $t2
lw $t3, 0($s3)	# load d to $t3
add $t0, $t1, $t2	# put b + c in $t0
sll $t0, $t0, 1	# Multiply b + c by 2
sll $t1, $t3, 2	# put 4*d in $t1
add $t0, $t1, $t0	# put 4*d + 2*(b+c) in $ t0
sw $t0, 0($s0)	# store the answer back in it's address ($s0)

Question 3 – MIPS to C

Consider the following Assembly MIPS code:

LOOP:	add $t0, $s0, $s1
	add $t1, $s0, $s2
	add $t0, $t0, $t1
	addi $t0, $t0, -3
	sll $t0, $t0, 3
	lw $t2, 0($s3)
	sub $t0, $t0, $t2
	sw $t0, 0($s4)

1. Assume that $s3 and $s4 contain the addresses of variables K,L. Translate the code to C.
2. Consider the same code with the line: “j LOOP” at the end. Translate the code with this modification.

Question 3 – Solution

1. Assume that $s3 and $s4 contain the addresses of variables K,L. Translate the code to C.

MIPS Assembly		C language
Label	Instruction	
LOOP:	add $t0, $s0, $s1	a = (b+c)+(b+d);
	add $t1, $s0, $s2	
	add $t0, $t0, $t1	
	addi $t0, $t0, -3	a = a-3;
	sll $t0, $t0, 3	a = a*8;
	lw $t2, 0($s3)	L[0] = a – K[0];
	sub $t0, $t0, $t2	
	sw $t0, 0($s4)	

Question 3 – Solution

2. Consider the same code with the line: “j LOOP” at the end. Translate the code with this modification.

MIPS Assembly		C language
Label	Instruction	

LOOP:	add $t0, $s0, $s1	while (true)
		{
		a = (b+c)+(b+d);

	add $t1, $s0, $s2	
	add $t0, $t0, $t1	
	addi $t0, $t0, -3	a = a-3;
	sll $t0, $t0, 3	a = a*8;
	lw $t2, 0($s3)	L[0] = a – K[0];
	sub $t0, $t0, $t2	
	sw $t0, 0($s4)	
	j LOOP	}

Question 4 – Branch & Loops

Consider the following C code (Assume i is an integer):

```
do {
       if (i==6) {
              i = i-2;
       }
       i--;
}
While (i>=0);
```

Assume i's <u>address</u> is located in $s0. Translate the code to MIPS assembly.

Question 4 – Solution

MIPS Assembly		
Label	Instruction	Comments
	addi $t2, $0, 6	# put the Constant 6 in $t2
	lw $t0, 0($s0)	# put i's value in $t0
LOOP:	bne $t0, $t2, Else	# if i ≠ 6, skip next two lines
	addi $t0, $t0, -2	# i = i-2
	sw $t0, 0($s0)	# Store i's new value in its proper address
Else:	addi $t0, $t0, -1	# Finished an iteration, decrease i by 1
	sw $t0, 0($s0)	
First option	slti $t1, $t0, 0	# if i<0, $t1=1, else $t1=0
	beq $t1, $0, LOOP	# if $t1=0, jump to LOOP
Second option	bgez $t0, LOOP	# if ($t0 ≥ 0) jump LOOP
Exit:	…	# loop broken, continue…

Question 5 – Logic Operations

1. Write an assembly code which inverts all the bits of $s0.
2. Translate the following Hexadecimal numbers to Decimal: 0xbadc0ffe, 0xba1ddeaf
3. Consider the following assembly operation:
   ```
   andi  $s0, $s0, 255
   ```
   Given that $s0 contains the value 0x287f2ad1, what word would be stored in $s0 after the execution of this instruction?
4. The instruction 0x00833020 is given in R-Format. Translate it to the ordinary MIPS assembly syntax.

Question 5 – Solution

1. Write an assembly code which inverts all the bits of $s0.

```
nor $s0, $s0, $s0
```

2. Translate the following Hexadecimal numbers to Decimal:

   0xbadc0ffe =

   $11 \cdot 16^7 + 10 \cdot 16^6 + 13 \cdot 10^5 + 12 \cdot 16^4 + 0 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16 + 14$
   $= 3{,}134{,}984{,}190$

   0xba1ddeaf =

   $11 \cdot 16^7 + 10 \cdot 16^6 + 1 \cdot 10^5 + 13 \cdot 16^4 + 13 \cdot 16^3 + 14 \cdot 16^2 + 10 \cdot 16 + 15$
   $= 3{,}122{,}519{,}727$

3. Consider the following assembly operation: `andi   $s0, $s0, 255`
   Given that $s0 contains the value 0x287f2ad1, what word would be stored in $s0 after the execution of this instruction?

   First, translate 0x287f2ad1 from Hex to bits, and 255 from decimal to bits:

0010	1000	0111	1111	0010	1010	1101	0001
0000	0000	0000	0000	0000	0000	1111	1111
`Bitwise And`							
0000	0000	0000	0000	0000	0000	1101	0001

Question 5 – Solution

4. The instruction 0x00833020 is given in R-Format. Translate it to the ordinary MIPS assembly syntax.

   To properly solve this question we need:

   - A list of opcode – Instruction pairs
   - A list of register number – register name pairs

   0x00833020 =

Opcode	rs	rt	rd	shamt	funct
000000	00100	00011	00110	00000	100000

```
add $a2, $a0, $v1
```

Signed & Unsigned

- In MIPS, negative numbers are represented in 2’s complement form.
- There are two sets of commands. One set for signed numbers and other are for unsigned:

Signed	Unsigned
`add $reg1, $reg2, $reg3`	`addu $reg1, $reg2, $reg3`
`addi $reg1, $reg2, $reg3`	`addiu $reg1, $reg2, $reg3`
`sub $reg1, $reg2, $reg3`	`subu $reg1, $reg2, $reg3`
`mul $reg1, $reg2, $reg3`	`mulu $reg1, $reg2, $reg3`
`slti $reg1, $reg2, const`	`sltiu $reg1, $reg2, const`

- Representing a result of an instruction performed on multiple numbers that requires more bits that are available is called “Overflow”
- Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as infinite number of digits to the left(the sign bit is incorrect: “0” on the left – when number is negative or a “1” when the number is positive.

Question 6 – Overflow

Assume we have registers of 4-bits only.
Add the following numbers and decide if there will be an overflow or not.

1. 1001, 1010 (unsigned)
2. 0101, 0111 (unsigned)
3. 0101, 0111 (signed)

Question 6 – Overflow

1. 1001, 1010 (unsigned)
   The answer is 10011_{10}, but notice we now need 5 bits to represent the result of the addition. Assuming we have only 4 bits - this is an **overflow.**
2. 0101, 0111 (unsigned)
   The answer is 1100_{10}, which equal 12. **No overflow** this time.
3. 0101, 0111 (signed)
   The answer is 1100_{10}. but this is a number in a 2's complement form, which is -4 (negate and add '1').
   This answer is not possible when adding two positive numbers. This is also an **Overflow**. Because we consider the numbers to be signed we have only 3 bits to represent the result (the fourth is the sign bit). 5+7 is impossible to represent using 3 bits. The largest positive number we can represent is 0111_{10}

Recitation 2 – Part B

Projects submission remarks

Revision of C

Revision of C Language

- Last week we discussed:
  - main()
  - Functions
  - Variables
  - printf
  - Arrays

- Today:
  - Multi-dimensional arrays
  - Strings
  - Keyboard input

Revision of C Language

Multi-Dimensional Arrays

- Can be initialized during declaration:
  - `int my_matrix [3][2] = { {1,0}, {0,1}, {1,1} };`
  - `Int zero_array[3][2] = { 0 };`

- But usually use a nested loop:
  - ```
    void init_array(int array[][100], int size, int value)
    {
       int i=0, j=0;
       for (i=0; i<size; i++) {
           for (j=0; j<100; j++) {
             array[i][j] = value;
           }
       }
    }
    ```

When passing an array to a function, you must specify all dimensions, beginning with the second
    ```

Revision of C Language

Strings

- A string is simply an array of characters (char) ending with the ‘\0’ character (zero in the ASCII table, also known as NULL).
- All the special string functions rely on the ‘\0’ character at the end.
- A string can be initialized during its declaration:
  - `char str[16] = "hello world";`
  - The double quotation marks tell the compiler to add the ‘\0’ character.

h	e	l	l	o		w	o	r	l	d	\0	?	?	?	?
str[0]											End of string				str[15]

- Note that ‘A’ and “A” are not the same. The single quotation mark is used for a simple character (no ‘\0’ added).
- Good practice- **always** assign strings with value before using them.

Revision of C Language

Strings

- The following function finds the length of a string. It relies on the existence of the '\0' character:

```
int my_strlen(char str[]) {
    int i=0;
    while (str[i]!='\0') {
        i++;
    }
    return i;
}
```

- What happens if we don't have the NULL character?
  - Possible options:
    - Infinite loop
    - Return different result each execution
    - Crash (if we access forbidden memory address)
  - Conclusion – don't do that

Revision of C Language

Keyboard Input

- We use the “scanf” function in a similar way to “printf”, in order to receive input from the user.
- Recall from last week, that except for arrays – variables passed to functions do not change. How can we then change the variables with the input? Use the ‘&’ symbol. We will discuss this when we talk about pointers.

- `scanf(“%d %lf”, &student_num, &average);`

- scanf ignores whitespaces. So the following are equivalent:
  ```
  scanf(“%d%d”,&i,&j);
  scanf(“%d %d”,&i,&j);
  ```
- If there is a non-whitespace character, then scanf expects to see it in the input stream. **If it doesn’t the function will fail**:
  ```
  scanf(“%d + %d”,&i,&j);
  ```

Revision of C Language

Keyboard Input

- If your input has more values than “scanf” expected, the rest will be kept for the following “scanf” functions.

- By clicking “Enter” after inputting through the keyboard, the “enter” will also be stored:
```
scanf(“%c”, &tav1);
scanf(“%c”, &tav2);
```
tav1 will hold ‘a’, and tav2 will hold ‘\n’.

- To avoid this, simply add ‘\n’ to your expected input format:
```
scanf(“%c”, &tav1);
scanf(“\n%c”, &tav2);
```
tav1 will hold ‘a’, and tav2 will hold ‘b’.

- You can also use the “getchar” function. The following are equivalent:
```
character = getchar();
scanf(“%c”, &character);
```

Revision of C Language

Keyboard Input

- Input of strings:
```
char answer[100];
scanf(“%s”, answer);
```
The input will continue up to the first whitespace or new line character. Note that we don’t add the ‘&’ character for strings. This will be explained later.

- The user is responsible for allocating enough space in the string, including the NULL character.

- You can also limit the number of character to read:
```
scanf(“%40s”, answer);
```