EN605.204.82.SU17 Computer Organization

Caleb Bowers

Assignment 3

2.3

For the C statement: `B[8] = A[i – j]` the MIPS assembly code looks as follows:

```
sub $t0, $s3, $s4  # Assign temp $t0 value i-j
sll $t0, $t0, 2    # Multiply i-j by 4 to get point in memory
lw $t1, 0($s6)     # Load temp $t1 with A[0]
add $t1, $t1, $t0  # Add the value of address i-j to 0 addr.
lw $t1, 0($t1)     # Load temp $t1 with value of A[i-j]
sw $t1, 32($s7)    # Store value $t1 into B[8]
```

2.7

Show how the value `0xabcdef12` would be stored in memory for both little-endian and big-endian machines:

Address	Little-Endian	Big-Endian
0	12	ab
4	ef	cd
8	cd	ef
12	ab	12

2.12

Assume:

`$s0 = 0x80000000`

`$s1 = 0xD0000000`

2.12.1

Calculate `$t0` for the following:

```
add $t0, $s0, $s1   # $t0 = $s0 + $s1
```

We need to first calculate the binary for both `$s0` and `$s1`:

`$s0 = 1000 0000 0000 0000 0000 0000 0000 0000`

`$s1 = 1101 0000 0000 0000 0000 0000 0000 0000`

Adding these together, we have:

`$t0 = 1 0101 0000 0000 0000 0000 0000 0000 0000 0000`

We can only store 32 bits in any given MIPS register, so

`$t0 = 0101 0000 0000 0000 0000 0000 0000 0000`

`$t0 = 0x50000000`

2.12.2

The result was not what we desired to calculate since overflow occurred as evidenced in 2.12.1 since we could not contain the entire number in the 32-bit register we were using to store the answer.

2.12.3

Calculate `$t0` for:

```
sub $t0, $s0, $s1  # $t0 = $s0 - $s1
```

We have the following binary values for our registers:

`$s0 = 1000 0000 0000 0000 0000 0000 0000 0000`

`$s1 = 1101 0000 0000 0000 0000 0000 0000 0000`

In order to calculate `$s0 - $s1` we can determine the two's complement of `$s1` and add it to `$s0`:

$\$s1_{\text{two's comp}}$ = 0011 0000 0000 0000 0000 0000 0000 0000

When we add this to `$s0` and store it in `$t0` we have:

`$t0 = 1011 0000 0000 0000 0000 0000 0000 0000`

`$t0 = 0xB0000000`

2.12.4

Overflow has not occurred since we did not exceed the 32-bit capacity of the register we are using to store the answer in. Two's complement accounts for the sign of the value in how it is store in memory, not by using a sign bit.

2.12.5

Calculate $t0 for the following MIPS assembly instructions:

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

From problem 2.12.1 we know that after `add $t0, $s0, $s1` `$t0` will equal:

`$t0 = 0101 0000 0000 0000 0000 0000 0000 0000`

`$t0 = 0x50000000`

Now to calculate `add $t0, $t0, $s0` we add `$s0` to our current value for `$t0`:

`$t0 = 0101 0000 0000 0000 0000 0000 0000 0000`

`$s0 = 1000 0000 0000 0000 0000 0000 0000 0000`

Doing so gives us:

`$t0 = 1101 0000 0000 0000 0000 0000 0000 0000`

`$t0 = 0xD0000000`

2.12.6

Overflow occurred in one step of our instruction set: in the first add. Therefore, the answer we get after the final instruction in register `$t0` is not accurate. It does not reflect the correct mathematical answer.

2.14

Provide the type and assembly language instruction for:

`0000 0010 0001 0000 1000 0000 0010 0000`

As a MIPS instruction this looks like:

	op	rs	rt	rd	shamt	funct
binary	000000	10000	10000	10000	00000	100000
decimal	0	16	16	16	0	32

Since the op is 0, the instruction is an R-Type. The function value is 32, which corresponds to the MIPS instruction `add`. All of the referenced registers are 16, which corresponds to register `$s0`. Combining these values into a MIPS instruction gives us:

```
add $s0, $s0, $s0  # $s0 = $s0 + $s0
```

2.15

Provide the type and hexadecimal representation of:

```
sw $t1, 32($t2)
```

The type of this instruction is an I-Type. The op section of the command does not equal zero

and there is a reference to an immediate value:

	op	rs	rt	Immediate Address
binary	101011	01010	01001	0000000000100000
decimal	43	10	9	32

From this, the binary for this instruction is:

```
1010 1101 0100 1001 0000 0000 0010 0000
```

Converting this to hexadecimal, we have:

```
0xAD490020
```

2.16

Provide the type, assembly language instruction, and binary representation described by:

```
op=0, rs=3, rt=2, rd=3, shamt=0, funct=34
```

The type of this instruction is an R-Type since `op` is zero. The binary representation is as

follows:

	op	rs	rt	rd	shamt	funct
binary	000000	00011	00010	00011	00000	100010
decimal	0	3	2	3	0	34

Binary command: `0000 0000 0110 0010 0001 1000 0010 0010`

This translates to the MIPS instruction:

```
sub $v1, $v1, $v0   # $v1 = $v1 - $v0
```

2.17

Provide the type, assembly language instruction, and binary representation of the following:

```
op=0x23, rs=1, rt=2, const=0x4
```

This type of instruction is an I-Type of instruction since there is a constant which will be used as an immediate value. The binary representation is as follows:

	op	rs	rt	Immediate Address
binary	100011	00001	00010	0000000000000100
decimal	35	1	2	4

Binary command: `1000 1100 0010 0010 0000 0000 0000 0100`

This translates to the MIPS instruction:

```
lw $v0, 4($at)   # Load 4 * value at $at in register $v0
```

2.19

Assume the following register contents:

```
$t0 = 0xAAAAAAAA, $t1 = 0x12345678
```

2.19.1

Find the value of `$t2` for the following:

```
sll $t2, $t0, 4
or $t2, $t2, $t1
```

In binary:

```
$t0 = 1010 1010 1010 1010 1010 1010 1010 1010
$t1 = 0001 0010 0011 0100 0101 0110 0111 1000

sll $t2, $t0, 4 :

$t2 = 1010 1010 1010 1010 1010 1010 1010 0000
```

```
or $t2, $t2, $t1 :

$t2 = 1010 1010 1010 1010 1010 1010 1010 0000
$t1 = 0001 0010 0011 0100 0101 0110 0111 1000
or
$t2 = 1011 1010 1011 1110 1111 1110 1111 1000   # binary
$t2 = 0x BABEFEF8   # hexadecimal
```

2.19.2

Determine the value of `$t2` for the following:

```
sll $t2, $t0, 4
andi $t2, $t2, -1
```

From problem 2.19.1 we know that `sll $t2, $t0, 4` results in `$t2` equalling:
`$t2 = 1010 1010 1010 1010 1010 1010 1010 0000`

So performing the next instruction gives us:

```
andi $t2, $t2, -1 :

$t2 = 1010 1010 1010 1010 1010 1010 1010 0000
-1  = 1111 1111 1111 1111 1111 1111 1111 1111
andi
$t2 = 1010 1010 1010 1010 1010 1010 1010 0000   # binary
$t2 = 0xAAAAAAA0   # hexadecimal
```

2.19.3

Determine the value of `$t2` for the following instructions:

```
srl $t2, $t0, 3
andi $t2, $t2, 0xFFEF


srl $t2, $t0, 3 :

$t0 = 1010 1010 1010 1010 1010 1010 1010 1010
srl 3
$t2 = 0001 0101 0101 0101 0101 0101 0101 0101
```

```
andi $t2, $t2, 0xFFEF :

$t2    = 0001 0101 0101 0101 0101 0101 0101 0101
0xFFEF = 0000 0000 0000 0000 1111 1111 1110 1111
andi
$t2    = 0000 0000 0000 0000 0101 0101 0100 0101   # binary
$t2    = 0x00005545   # hexadecimal
```