Ira A. Fulton Schools of Engineering
School of Computing, Informatics and Decision Systems Engineering

SER 450 **COMPUTER ARCHITECTURE**

Your Name: **Jacob Hreshchyshyn** Jacob Hreshchyshyn

When completing this worksheet, please use a different color text or typeface so that we can easily identify your work. Remember to show your work / give justification for your answers.

PRACTICE PROBLEMS 2

Problem 1:

Provide the type and assembly language instruction for the following hexadecimal value: 0x02108020

One approach for this solution is to convert this hexadecimal value into a binary value, then determine the opcode from the binary value, which will help determine the rest of the instruction.

0 in hex becomes 0000 in binary.

2 in hex becomes 0010 in binary.

1 in hex becomes 0001 in binary.

0 in hex becomes 0000 in binary.

8 in hex becomes 1000 in binary.

0 in hex becomes 0000 in binary.

2 in hex becomes 0010 in binary.

0 in hex becomes 0000 in binary.

So, the resulting binary number is 0000 0010 0001 0000 1000 0000 0010 0000.

Based on the hex value beginning with 0, we know that this is an R-Type instruction, so we can rearrange this binary value based on the bit format we expect.

000000 10000 10000 10000 00000 100000

6 bits 5 bits 5 bits 5 bits 5 bits 6 bits

Given that this binary number represents an R-Type instruction and that the 6 bits at the end of the number can be represented as 20 in hex, it is clear that this is an addition instruction. There is no shift amount, as shown by the penultimate collection of bits numbering 5. Now we can examine the first source register, the second source register, and the destination register, all of

which are represented as 10000, which is 16 in decimal. According to the green sheet, 16 is the first register $s0, which is used as a Saved Temporary.

Thus, the assembly language instruction would be the following: add $s0, $s0, $s0

Problem 2:

Provide the type and hexadecimal representation of the following instruction: `sw $t1,32($t2)`

First, we will represent this as in the table in page 85 of the textbook.

43	10	9		32	
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The above represents the decimal values associated with the opcode, rs, rt, and address. We can glean from the example in page 85 that the opcode is 43, but we also know this from the function associated with it in the green sheet, which is 2b in hex, which is equivalent to 43. 10 is the decimal value of our rs, and 9 is the decimal value of our rt, leaving 32 as the decimal value of our address.

Converting each to binary and maintaining the I-Type form (which is given away by the store word instruction), we get the following:

101011	01010	01001	00000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

We can then group this binary number into collections of 4 bits and convert those to hexadecimal.

1010	1101	0100	1001	0000	0000	0010	0000
A	D	4	9	0	0	2	0

Thus, the final hexadecimal instruction from the I-Type instruction above is 0xAD490020.

Problem 3:

Assume that we would like to expand the MIPS register file to 128 registers and expand the instruction set to contain four times as many instructions.

How could each of the two proposed changes decrease the size of a MIPS assembly program?

Because MIPS is currently limited to 32 registers, increasing the register file to 128 can decrease the size of a MIPS assembly program by providing the programmer with more resources for storing data, which can mitigate the need for additional instructions to effectively reuse registers.

Similarly, if there were four times as many instructions as there are currently, this could accommodate other operations that are not directly available by the existing set of instructions. For example, while there exist Set Less Than instructions, there are no explicitly defined Set Greater Than instructions. The existence of such instructions would minimize the size of a MIPS

assembly program by eliminating the need to implement behaviors that do not exist within the existing instruction set.

How could each of the two proposed changes increase the size of a MIPS assembly program?

Both changes can increase the size of a MIPS assembly program through the inadvertent encouragement of lazy programming. For example, increasing the number of registers can encourage developers to use registers less conservatively and more inefficiently.

Similarly, increasing the instruction set to include four times as many instructions can simply increase the complexity of the assembly program, which can lead either to sub-optimal use of the new instructions where other instructions would have sufficed or to slower programs since the new instructions with their potential complexity might be slower to execute.

Problem 4:

Translate the following loop into C. Assume that the C-level integer I is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of array MemArray.

```
      addi $t1, $0, 0
LOOP: lw   $s1, 0($s0)
      add  $s2, $s2, $s1
      addi $s0, $s0, 4
      addi $t1, $t1, 1
      slti $t2, $t1, 100
      bne  $t2, $0, LOOP
```

Integer I held in $t1

Integer result held in $s2

Base address of array MemArray stored in $s0

The resulting C code would be the following:

```
for (I = 0; I < 100; I++) {

        result = result + *MemArray;

        MemArray = MemArray + 1;

}
```

The first instruction initializes $t1, known to be the C-level integer I, to 0. This is done in the first instruction in our for loop parentheses set.

The second instruction marked by LOOP gets the word or integer stored in the address $s0. This is equivalent to our dereferencing instruction *MemArray.

The third instruction adds the accessed value of MemArray to $s2, which represents the result, and stores that result in $s2. This third instruction helps fully represent what is happening in result = result + *MemArray;.

The fourth instruction then adds 4 to $s0, which is equivalent to shifting the base address to access the next word in the memory array. This, in C, is pointer manipulation and is demonstrated in the instruction MemArray = MemArray + 1.

The fifth instruction adds 1 to $t1 and stores that result in $t1, which is equivalent to I++.

The last two instructions give us the conditional nature of our for loop. The first of the two uses the Set Less Than Immediate instruction to determine if $t1 (or I) is less than 100. The second of the two instructions then jumps back to the LOOP portion of the code if the result of the Set Less Than Immediate instruction, stored in $t2, is not equal to 0. It does this because the Set Less Than Immediate instruction is set to 1 if the value being measured is less than the compared value. So, if the value in $t1 is less than 100, $t2 will store a 1, which will trigger the bne jump back to LOOP. Otherwise, if the value in $t1 is equal or greater than 100, $t2 will store a 0 and no jump will occur. The first of the two instructions represents the I < 100 and the second of the two instructions represents the implicit jump back to the beginning of the loop.

Problem 5:

Functions can often be implemented by compilers "in-line". An in-line function is when the body of the function is copied into the program space, allowing the overhead of the function call to be eliminated. If an "in-line" version of the C code below is implemented in MIPS assembly, what is the total reduction in MIPS instructions?

```
int fib(int n) {
   if (n==0) return 0;
   else if (n == 1) return 1;
   return fib(n-1) + fib(n-2);
}
```

The issue with attempting to in-line this fib function is that it is recursive. If a compiler attempted to in-line the body of the function where the function is called within the function itself, this will expand the program infinitely. There would be no reduction in MIPS instructions.

Problem 6:

Assume for a given processor the CPI of arithmetic instructions is 1, the CPI of load/store operations is 10, and the CPI of branch instructions is 3. Assume a program has the following instruction breakdowns: 500 million arithmetic instructions, 300 million load/store instructions, 100 million branch instructions.

Suppose that a new processor becomes available that has more powerful arithmetic instructions. On average, the new instructions would reduce the number of arithmetic instructions of the original code by 25% but the clock cycle time will increase by 10%. Based solely on performance, should we move our code to the new processor?

First, we will assume that the only changes between the new processor and the old processor are that the arithmetic instructions are reduced by 25% and that the clock cycle time increases by 10%.

We begin by finding the total instruction count of the first processor.
500,000,000 arithmetic instructions + 300,000,000 load/store instructions + 100,000,000 branch instructions = 900,000,000 total instructions.

Next, we can find the number of clock cycles for this processor by multiplying each class's CPI with their corresponding instruction count and summing each product.

Clock Cycles = (500,000,000 * 1) + (300,000,000 * 10) + (100,000,000 * 3) = 3,800,000,000.

While this next step might be redundant, my process led me to calculate the Average CPI of this processor. This is done by dividing the Clock Cycles by the earlier discovered total processor instruction count.

CPI = 3,800,000,000 / 900,000,000 = 4.2

Next, we can determine the Clock Cycles and the CPI of the new processor using the earlier described assumptions.

First, before finding the total instruction count of the new processor, we must determine the instruction count of the arithmetic instructions, which is reduced by 25% in this processor.

500,000,000 – (500,000,000 * (25/100)) = 375,000,000

Next, we can find the total instruction count of the new processor.

375,000,000 arithmetic instructions + 300,000,000 load/store instructions + 100,000,000 branch instructions = 775,000,000 total instructions.

Next, we can find the number of clock cycles for this processor as done before.

Clock Cycles = (375,000,000 * 1) + (300,000,000 * 10) + (100,000,000 * 3) = 3,675,000,000.

Now we can find the average CPI of this new processor as done before.

CPI = 3,675,000,000 / 775,000,000 = 4.74

Now we can attempt to find the speedup of the new processor over the old processor by dividing the CPU Time of the old processor by the CPU Time of the new processor. With the information we have found so far, we can use the model CPU Time = Instruction Count * CPI * Clock Cycle Time to help us make this comparison.

CPU Time/New CPU Time = (900,000,000 * 4.2 * Clock Cycle Time)/(775,000,000 * 4.74 * (Clock Cycle Time + (Clock Cycle Time * 0.1)))

The Clock Cycle Time in the denominator is represented this way because the problem states that the new processor's Clock Cycle Time is increased by 10% compared to the old processor. The Clock Cycle Time terms must be cancelled out, so we can address this portion of the product first.

Clock Cycle Time / (Clock Cycle Time + (Clock Cycle Time * 0.1)) =
Clock Cycle Time / (Clock Cycle Time + (Clock Cycle Time / 10)) =
Clock Cycle Time / ((10 Clock Cycle Time + Clock Cycle Time) / 10) =

Clock Cycle Time / ((11 Clock Cycle Time) / 10) =
Clock Cycle Time * (10/11 Clock Cycle Time) =
10/11

So, we now have the following:

CPU Time/New CPU Time = (900,000,000 * 4.2 * 10)/(775,000,000 * 4.74 * 11) = 0.94

This result is less than 1, which tells us that there is no performance increase when comparing the old processor with the new processor. Therefore, we should not move our code to the new processor if we are concerned about performance.