Spring 2024
Computer Architecture
Homework 02

Max points: 100

---

1. (15 points) You are given with following register values:
   x5 = $0x00000000AAAAAAAA$
   x6 = $0x1234567812345678$

   (a) For the register values shown above, what is the value of x6 for the following sequence of instructions? (5 points)

```
    and x7, x5, x6
    srli x7, x7, 16
    addi x8, x0, 0x21F
    blt x8, x7, ELSE
    beq x0, x0, IF
ELSE: ori x6, x0, 2
    beq x0, x0, EXIT
IF: andi x6, x0, 2
EXIT:
```

Solution:
1. x7 = 0x2200228
2. x7 = 0x220
3. x8 = 0x21f
4. x8 < x7 −− > True, go to ELSE
5. **x6 = 2**

(b) For the register values shown above, what is the value of x7 for the following sequence of instructions? (5 points)

```
slli x7, x6, 4
andi x5, x7, 0
or x7 x7 x5
```

Solution:
x7 = $0x2345678123456780$

(c) For the register values shown above, what is the value of x7 for the following sequence of instructions? (5 points)

```
srli x7, x5, 3
slli x7, x7 2
ori x7 x7 0
```

Solution:

x7 = $0x55555554$

2. (10 points) Translate the following C code to RISC-V assembly code. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the double array K. Explain your code by writing explanatory comments

```
\\ int64_t corresponds to a double integer.
int64_t leaf_procedure(int64_t a, int64_t b, int64_t i,int64_t j, int64_t K[]) {
    int64_t temp = a + b;
    int64_t temp2 = i + j;
    if (a > b) {
        if (temp > temp2)  {
            K[i] = b;
            }
        else {
            K[i] = a;
        }
    }

    else {
        K[j] = temp;

     }

}
```

Solution:

```
LEAF: addi sp,sp,-24 // sp is the stack pointer
sd x11 16(sp)
sd x8, 8(sp)
sd x9, 0(sp) // reserve 2 temporary registers.


add x8, x5, x6 // x8 = a+b
add x9, x7, x29 // x9 = i+j
blt x6,x5, IF // if a>b, then go to IF


slli x11, x29,3 // else block
add x11, x11, x10 // x9 is a temp register to store the offset.
sd x8, 0(x11) // K[j] = temp
beq x0,x0, END // go to END


IF: blt x9, x8, IF_1 # if temp> temp2, go to IF_1

slli x11, x7, 3
add x11, x11, x10
sd x5, 0(x11) // K[i] = a
beq x0,x0, END // go to END
```

```
IF_1:
slli x11, x7, 3
add x11, x11, x10
sd x6, 0(x11) // K[i] = b


END: ld x9, 0(sp)
ld x8, 8(sp)
ld x11, 16(sp) // restore the values of the temporary registers
addi sp,sp,24 // restore the stack pointer
jalr x0, 0(x1) // return to the caller
```

3. (15 points) Consider the following RISC-V loop:

```
LOOP: beq x6, x0, DONE
    addi x6, x6, -1
    addi x5, x5, 2
    jal x0, LOOP
DONE:
```

(a) For the loop written in RISC-V assembly above, assume that the register x6 is initialized to the value N. How many RISC-V instructions are executed? (5 points)

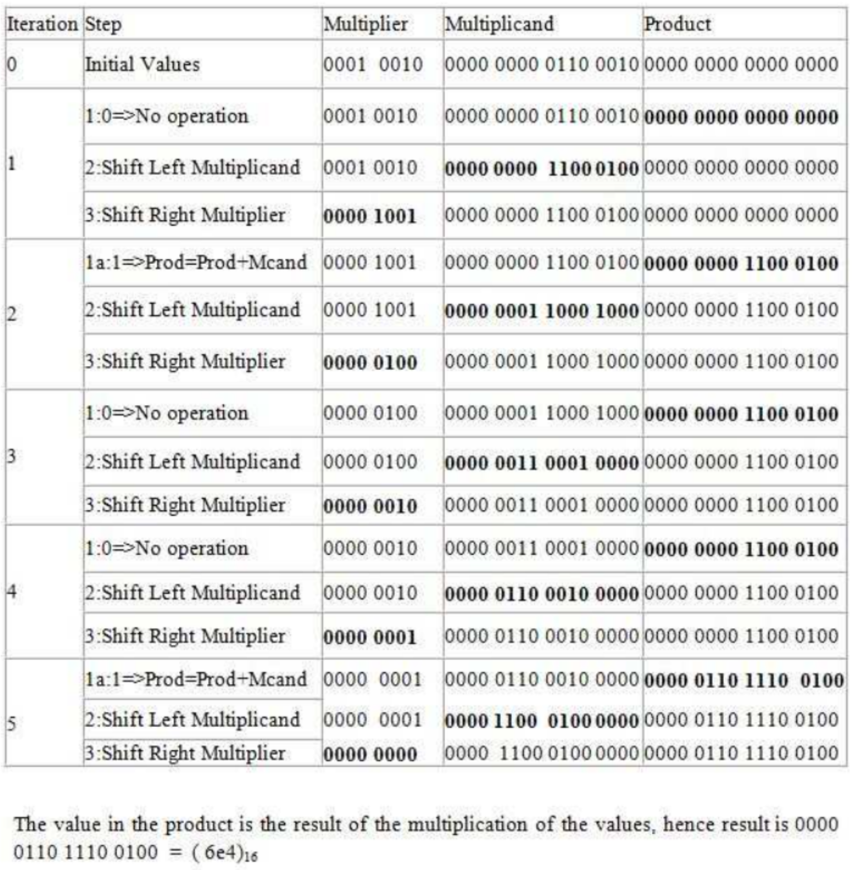
Solution: 4*N + 1 instructions.

(b) For the loop written in RISC-V assembly above, replace the instruction "beq x6, x0, DONE" with the instruction "blt x6, x0, DONE" and write the equivalent C code. (10 points)

Solution: (Note: change condition ! = to ¿ = in the while loop)

```
    acc = 0;
    i = 10;
    while (i >= 0) {
     acc += 2;
     i--;
    }
```

4. (10 points) Using a table similar to that shown in Figure 3.6, calculate the product of the hexadecimal unsigned 8-bit integers 62 and 12 using the hardware described in Figure 3.5. You should show the contents of each register on each step.



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial Values	0001 0010	0000 0000 0110 0010	0000 0000 0000 0000
	1:0=>No operation	0001 0010	0000 0000 0110 0010	**0000 0000 0000 0000**
1	2:Shift Left Multiplicand	0001 0010	**0000 0000 1100 0100**	0000 0000 0000 0000
	3:Shift Right Multiplier	**0000 1001**	0000 0000 1100 0100	0000 0000 0000 0000
	1a:1=>Prod=Prod+Mcand	0000 1001	0000 0000 1100 0100	**0000 0000 1100 0100**
2	2:Shift Left Multiplicand	0000 1001	**0000 0001 1000 1000**	0000 0000 1100 0100
	3:Shift Right Multiplier	**0000 0100**	0000 0001 1000 1000	0000 0000 1100 0100
	1:0=>No operation	0000 0100	0000 0001 1000 1000	**0000 0000 1100 0100**
3	2:Shift Left Multiplicand	0000 0100	**0000 0011 0001 0000**	0000 0000 1100 0100
	3:Shift Right Multiplier	**0000 0010**	0000 0011 0001 0000	0000 0000 1100 0100
	1:0=>No operation	0000 0010	0000 0011 0001 0000	**0000 0000 1100 0100**
4	2:Shift Left Multiplicand	0000 0010	**0000 0110 0010 0000**	0000 0000 1100 0100
	3:Shift Right Multiplier	**0000 0001**	0000 0110 0010 0000	0000 0000 1100 0100
	1a:1=>Prod=Prod+Mcand	0000 0001	0000 0110 0010 0000	**0000 0110 1110 0100**
5	2:Shift Left Multiplicand	0000 0001	**0000 1100 0100 0000**	0000 0110 1110 0100
	3:Shift Right Multiplier	**0000 0000**	0000 1100 0100 0000	0000 0110 1110 0100

The value in the product is the result of the multiplication of the values, hence result is 0000 0110 1110 0100 = $(6e4)_{16}$

Figure 1: Solution Q4

5. (10 points) Assume that the two parameters array and size are found in the registers x10 and x11, and that i is allocated to register x5 for array version and p is allocated to register x5 for pointer version.
(a) Convert the following C code which uses arrays and array indices to RISC-V assembly code.

```
initialize1(long long int array[], size_t int size)
{
size_t i;

for (i = 0; i < size; i += 1)

array[i] = 10;
}
```

Solution: Assuming that x12 contains 10

```
li x5, 0 // i = 0
loop1: slli x6, x5, 3 // x6 = i * 8
add x7, x10, x6 // x7 = address of array[i]
sd x12, 0(x7) // array[i] = 10
addi x5, x5, 1 // i = i + 1
blt x5, x11, loop1 // if (i < size) go to loop1
```

(b) Convert the following C code which uses pointers to RISC-V assembly code.

```
initialize2(long long int *array, size_t int size)
{
long long int *p;

for (p = &array[0]; p < &array[size]; p = p + 1)

*p = 10;
}
```

Solution: Assuming that x12 contains 10

```
mv x5, x10 // p = address of array[0]
slli x6, x11, 3 // x6 = size * 8
add x7, x10, x6 // x7 = address of array[size]
loop2: sd x12, 0(x5) // Memory[p] = 10
addi x5, x5, 8 // p = p + 8
bltu x5, x7, loop2 // if (p < &array[size]) go to loop2
```

6. (5 points) Compare the performance of the array version of the program given in Question 5(a) with the pointer version of the program given in Question 5(b). Which version executes more instructions per iteration?

Solution:

Comparing the two code sequences illustrates the difference between array indices and pointers:

The array version of the program must have the "multiply" and add inside the loop because i is incremented and each address must be recalculated from the new index. The memory pointer version of the program increments the pointer p directly.

The pointer version moves the scaling shift and the array bound addition outside the loop, thereby reducing the instructions executed per iteration from five to three. (As a side note: Remember that this manual optimization corresponds to the compiler optimization of strength reduction (shift instead of multiply) and induction variable elimination (eliminating array address calculations within loops).

7. (5 points) If you have a-bit multiplicand and b-bit multiplier, how many bits you need to represent the product. Give explanation.

Solution: The product of an a-bit multiplicand and a b-bit multiplier can have at most a+b bits.

To see why this is the case, consider the multiplication process. When multiplying a digit of the multiplicand by a digit of the multiplier, the result can be at most the sum of the number of digits in each of the multiplicand and multiplier. For example, when multiplying a 3-digit number by a 2-digit number, the largest possible product is 6 digits long.

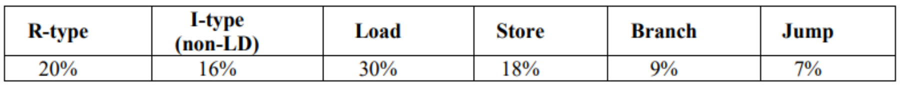
Therefore, to represent the product of an a-bit multiplicand and a b-bit multiplier, you need at least a+b bits. This ensures that you have enough space to represent the largest possible product. However, the product may not always require all a+b bits, and the actual number of bits required will depend on the specific values of the multiplicand and multiplier.

8. (5 points) Given two numbers X = 185 and Y = 122, answer the following questions. Show your working.
(a) Assume X and Y are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 − 122. Is there overflow, underflow, or neither?
(b) Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 + 122. Is there overflow, underflow, or neither?

Solution:

a) Overflow (-57 - 122 = -179), which does not fit into an SM 8-bit format)
b) Neither (-57 + 122 = 65)

9. (10 points) Consider the following instruction mix



R-type	I-type (non-LD)	Load	Store	Branch	Jump
20%	16%	30%	18%	9%	7%

Figure 2: Instruction Mix

(a) Assume that the data memory fails. What fraction of instructions will still execute successfully? (assume all instructions are independent of each other) (2.5 points)

Solution:

Load and store use data memory which accounts for 30 + 18 = 48 % instructions.
If data memory fails then 100 − 48 = 52% instructions execute successfully.

(b) Assume instruction memory fails. What fraction of instructions will execute successfully. (assume independence)? (2.5 points)

Solution:

0% since instruction memory is required by all instructions.

(c) Bob is redesigning the RISC-V architecture to make it power efficient. He has identified that there are some instructions where immediate data generation is useless and simply ignored. In what case(s) is immediate data generation useless. Suggest a high level modification in the design so that immediate data is not generated in the cases where it is not required. (5 points)

Solution:

Immediate data generation is necessary for every instruction except R-type. The given architecture is designed such that the immediate generation always takes place but is ignored when it is not needed through MUXs and control signals.

One possible solution

We can introduce an additional component in the architecture which checks if the current instruction is an R-type instruction through its opcode. Moreover, we can redesign the ImmGen unit such that it has an on/off bit. Now if the current instruction is an R-type instruction, the additional unit produces a signal which is then passed to the Imm Gen unit to turn it off.

10. (15 points) Given the RISC V processor implementation below, answer the following:

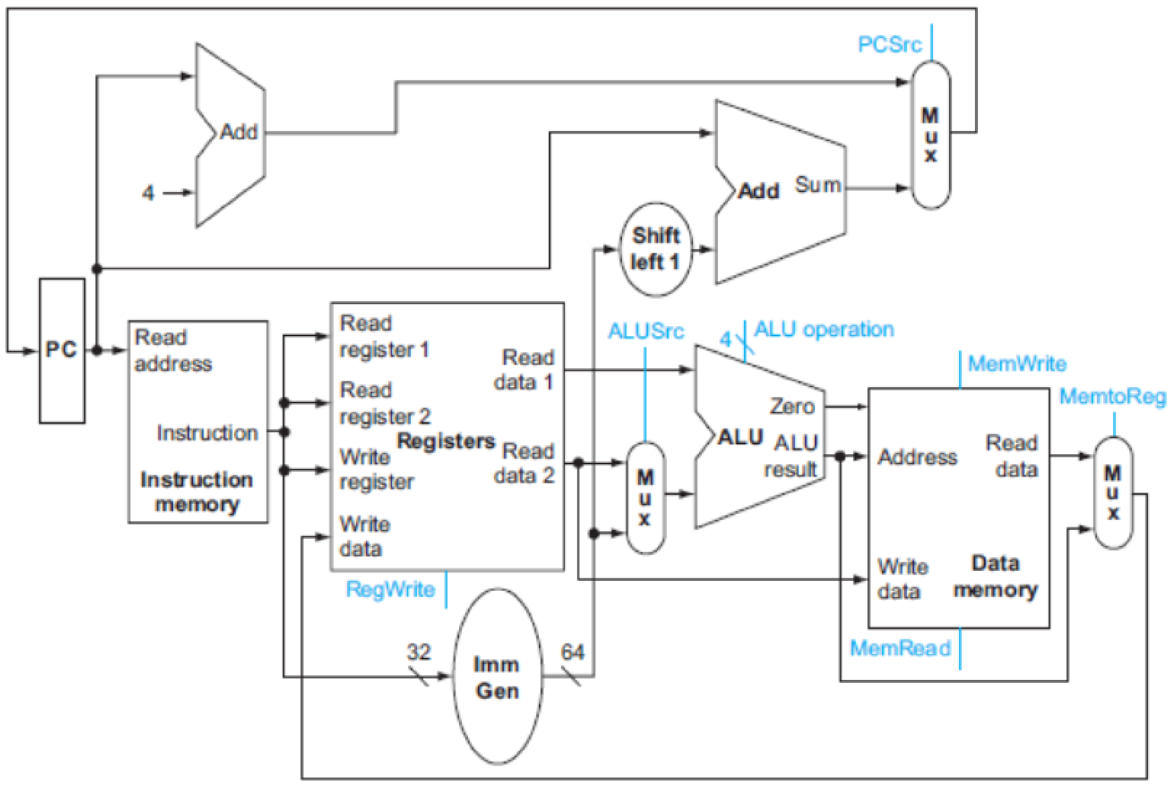


Figure 3: RISC-V architecture

a) For the instructions below, identify the components of the processor it uses and what are the values of MemRead, MemWrite, RegWrite, and MemtoReg. (10 points)

i) add x3,x4,x5
ii) sd x29,200(x2)
iii) ld x8,64(x2)

iv) addi x5,x6,-5

Solution: In every instrcution, the Program Counter, Instruction Memory and Register File is used. The additional components used for each instrcution are.
i) 4-Adder, ALU, ALUSrc Mux, MemtoReg Mux
ii) : 4-Adder, ALU, ImmGen, ALUSrc Mux, MemtoReg Mux, Data Memory
iii) 4-Adder, ALU, ImmGen, ALUSrc Mux, Data Memory
iv) : 4-Adder, ALU, ImmGen, ALUSrc Mux, MemtoReg Mux

Intruction	MemRead	MemWrite	RegWrite	MemtoReg
add x3,x4,x5	0	0	1	0
sd x29,200(x2)	0	1	0	x
ld x8,64(x2)	1	0	1	1
addi x5,x6,-5	0	0	1	0

(b) To support branch instructions, what must be added to the processor shown above? What other components of the processor are used when a branch instruction is being run and why are they used? (2 points)

Solution:

Control unit / Branch signal Branch signal is used to identify if the instruction passed is a branch signal. It is used as an input to the and gate described below.

And gate

To support branch instructions, an 'and' gate should be introduced between Control Units' branch signal and the ALU's zero signal. This will pass the output of the 'and' gate to the ALUSrc signal of the multiplexer which passes next address to Program Counter. This and gate will ensure that the program only jumps to the given label if and only if the the branch condition comes out to be true from the ALU. If the branch condition is false, then the and gate would not allow the multiplexer to allow jumping to label but simply move on to the next instruction in the instruction memory.

Other components

The adder is used to add the offset to fetch the instruction that a branch points to incase the branch condition is true.

The multiplexer is used to select the source of the offset to fetch the next instruction. Based on the PCSrc signal, it outputs the offset from either the 4-Adder or branch adder.

(c) A RISC-V based embedded system was sent to space having the same architecture as above. Due to interference of cosmic rays, the PC adder has malfunctioned and it now adds 8 instead of 4. Explain how exactly will the malfunction affect the program being executed on the system. (3 points)

Solution: In the given architecture, due to the size of the one instruction, we add 4 to the PC to execute the next instruction. If the adder adds 8 to the previous offset then the counter would skip one instruction and jump to the next instruction. So in case of the program given above instruction 1 and 3 will be executed and 2,4 will be skipped.