

# Computer Architecture

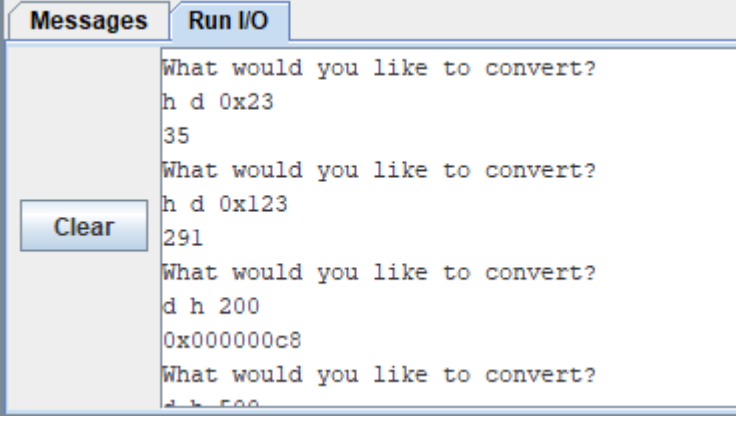
## Assignment 1

### Isa and RISC-V Programming

#### Problem1:

In this problem, each code is implemented on the RARS simulator and all the system calls of RARS are used which can be found on the [link](#). All the sub-problems are provided in separate files.

A.

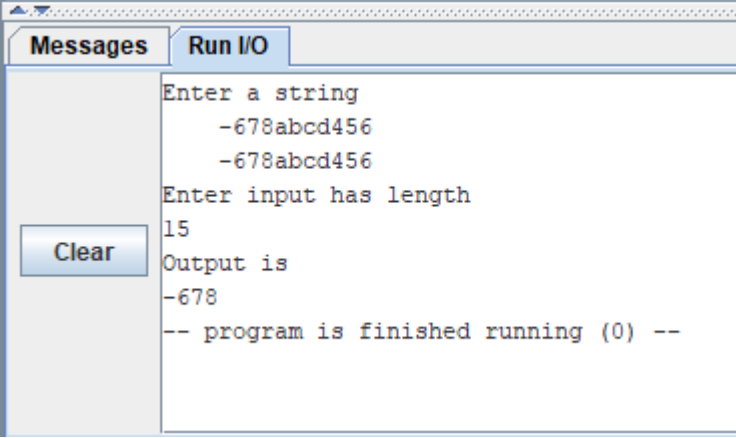


The screenshot shows the 'Messages' tab of the RARS simulator. The text in the window is as follows:

```
What would you like to convert?  
h d 0x23  
35  
What would you like to convert?  
h d 0x123  
291  
What would you like to convert?  
d h 200  
0x000000c8  
What would you like to convert?  
d h 500
```

A 'Clear' button is visible on the left side of the window.

B.

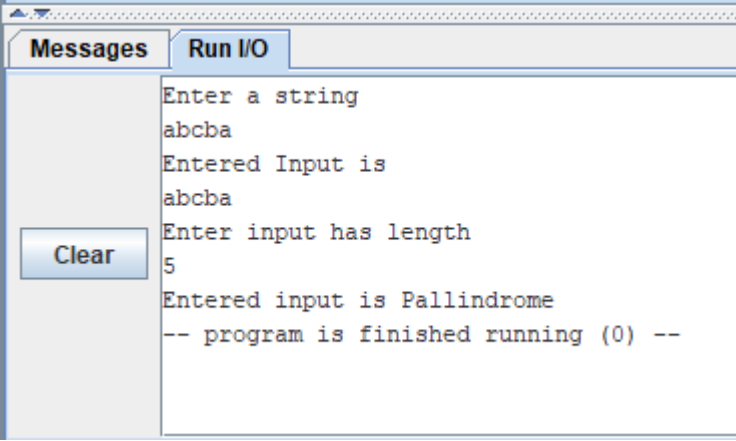


The screenshot shows the 'Messages' tab of the RARS simulator. The text in the window is as follows:

```
Enter a string  
-678abcd456  
-678abcd456  
Enter input has length  
15  
Output is  
-678  
-- program is finished running (0) --
```

A 'Clear' button is visible on the left side of the window.

C.



The screenshot shows the 'Messages' tab of the RARS simulator. The text in the window is as follows:

```
Enter a string  
abcba  
Entered Input is  
abcba  
Enter input has length  
5  
Entered input is Pallindrome  
-- program is finished running (0) --
```

A 'Clear' button is visible on the left side of the window.



**Problem 2:**

A.

ISA defines the interface between the software and hardware that is the interaction between the software and the hardware. Microarchitecture deals with the implementation of hardware. This distinction allows the software to truly stay independent of hardware. As a programmer, you do not have to worry about the underlying architecture. Optimizations and modifications can be made at the uarch without affecting the legacy software which was developed earlier.

B.

The compiler does not need to know about uarch. Only knowledge of ISA is sufficient since the compiler will be generating the ISA (or binary). However, knowledge of uarch is important since as a programmer you can help the compiler to do optimizations which the compiler will always try to do. Like loop unrolling, register usage (may lead to code spill), out-of-order execution, branch prediction  
Example: Loop unrolling - in below example loop is unrolled to reduce computations

```
while(i<10)
{
    x[i]=0;
    i++;
}
```

```
while(i<10)
{
    x[i]=0;
    i++;
    x[i]=0;
    i++;
}
```

**Problem3:**

- a. Instructions are 32 bits wide: **ISA**
- b. Instructions are always executed in order. : **uarch**
- c. The ALU does not have a subtraction module: **uarch**
- d. There is no multiplication instruction: **ISA**
- e. The MAR and MDR registers are used for memory reads and writing. : **uarch**
- f. There are 12 general-purpose registers that instructions can use: **ISA**
- g. There are three condition codes (n, z, and p) representing the result of the previous instruction. : **ISA**
- h. It takes 6 cycles to execute a multiplication instruction. : **uarch**
- i. Instructions are pipelined in four stages: **uarch**

**Problem4:**

- a. Give one use case for a NOP instruction: **To insert delays**
- b. Say RISC-V does not provide a NOP opcode. How can we simulate one with the existing instruction set? Provide the machine code for this instruction.

Sol: **we can use add x0,x0,x0 ad no operation instruction.**

**Problem5:**

- a. Is this decision an ISA or microarchitecture design specification? **ISA**
- b. Which of these possibilities support subroutine nesting (i.e. subroutine calls another subroutine)?

**Both 2 and 3**

- c. Which of these possibilities support recursion (i.e. subroutine calls itself)? **Only 3**

**Problem6:**

At address x1000 and x1001, x12 and x42 are stored now if the machine (16 bits) is:

1. Little-endian then byte[0]=12 and byte[1]=42 complete 16 bits is x4212.  
2's complemented is x4212 since sign bit is 0.
2. Big=-endian then byte[0]=42 and byte[1]=12 complete 16 bits is x1242.  
2's complemented is x1242 since sign bit is 0.

At address x1002 and x1003, x32 and x21 are stored now if the machine (16 bits) is:

3. Little-endian then byte[0]=32 and byte[1]=21 complete 16 bits is x2132..  
2's complemented is x2132 since sign bit is 0.
4. Big=-endian then byte[0]=21 and byte[1]=32 complete 16 bits is x3221.  
2's complemented is x3221 since sign bit is 0.

**Problem7:**

1. the ISA is bit-addressable -  $64\text{MB} = 64 * 8\text{Mb/b} = 512\text{M} = 29 \text{ bits}$
2. the ISA is byte-addressable -  $64\text{MB}/8\text{b} = 8\text{M} = 23\text{bits}$
3. the ISA is 128-bit addressable -  $512\text{Mb}/128\text{b} = 4\text{M} = 22 \text{ bits}$

**Problem8:**

1.

A . In zero address machine(using only stack)

PUSH B

PUSH C

MUL

PUSH A

ADD

PUSH D

PUSH C

MUL

PUSH E

ADD

PUSH D

SUB

MUL

POP X

B. In one address machine.

LOAD C (ACC = C)

MUL B (B = B\*ACC)

LOAD B (ACC = B\*C)

ADD A (A = A + ACC)

LOAD D (ACC = D)

MUL C ( $C = C * ACC$ )  
 LOAD C ( $ACC = D * C$ )  
 ADD E ( $E = E + ACC$ )  
 LOAD D ( $ACC = D$ )  
 SUB E ( $E = D - ACC$ )  
 LOAD A ( $ACC = A + (B * C)$ )  
 MUL E ( $E = D - (E + (D * C))$ )  
 LOAD E  
 ADD X ( $X = X + E$ )

C. Two address instructions

MUL B,C ( $B = B * C$ )  
 ADD A,B ( $A = A + B * C$ )  
 MUL C,D ( $C = C * D$ )  
 ADD E,C ( $E = E + C * D$ )  
 SUB D,E ( $D = D - E + C * D$ )  
 ADD A,D  
 ADD X,A

D. In Three address machine

MUL B,B,C  
 ADD A,A,B  
 MUL C,C,D  
 ADD E,E,C  
 SUB D,D,E  
 MUL A,A,D  
 MV X,A

2.

Advantage: The number of instructions in one address is less as compared to zero addresses.

Disadvantage: The time taken by one address is more compared to zero address since instruction length is longer than zero address.

### Problem9:

1.

No. of bits required for opcodes = 3 (as  $2^3 = 8$ ).

The total size of an instruction =  $3 + 5 = 8$  bits

Number of addresses =  $2^8 = 256$  addresses.

2. For any general-purpose register, 2 bits are required. So 4 general-purpose registers are present.

3. For MOV R2, R1 we need to copy the value of register R1 to R2.

It can be done as

LI 0 (R0 = 0)  
 AND R2,0,R0 ( $R2 = R2 \& 0 = 0$ )  
 ADD R2,0,R1 ( $R2 = R2 + R1 = R1$ )  
 AND R1,0,R0 ( $R1 = R1 \& 0 = 0$ )

**Problem10:**

2.

The jump-and-link-register instruction (JALR) is the union of JAL and JR, meaning that it transfers control to the address in a specified register which is provided by the user, and stores the return address in the register file which can be the same. However, unlike JAL, JALR allows the programmer to specify the destination register of the return address.

Hence it is useful in situations where the return address is needed like function call, etc.

**Problem11:**

1.

The above program does not work as intended because the return address is not provided in the j instruction, thus after the ret statement in the subroutine is called it does not know where to return.

2.

3.

In subroutine, we can change the return to another jump and add that label as A3.

```
MAIN2 LA t3, L2
```

```
A2 J t3
```

```
A3 addi a0,x0,10
```

```
ecall
```

```
L2 ADD t2, t1, x0
```

```
J A3
```

**Problem12:**

The tradeoff can be explained using the difference between Fixed and Variable length.

S.No.	Fixed Length	Variable Length
1.	Less number of instructions.	More number of instructions
2.	Approximately one cycle for each instruction.	More than one cycle for instructions.
3.	Less numbers of registers and addressing modes.	More number of registers and addressing modes
4.	Simple instructions	Complex instructions

Fixed length can be under RISC and variable length can be under CISC.

At the hardware level we can say that there would be more cycles hence more computation time as well as complex circuits in variable length.

At software level we can say that the user can perform complex operations easily.

**Problem13:**

1.

After the completion entries can be (Based on assembly described in 2).

Register	Before	After
t0	4	4
t1	5	5
t2	1	1
t3	0	18
t4	10	10
t5	0	1
t6	2	0
t7	0	9

2.

Based on defined assembly the value in table is filled.

ADD T3,T0,T1

ADD T3,T3,T3

ADD T5,T5,T2

ADD T7,T0,T1

3.

Let the address locations of various registers as defined below.

T0 - 00000

T1 - 00001

T2 - 00010

T3 - 00011

T4 - 00100

T5 - 00101

T6 - 00110

T7 - 00111

Machine instructions of above defined instructions are based on the format of add instruction defined in green sheet.

I - funct7 sd2 sd1 funct3 rd opcode

I1 - 0000000 00001 00000 000 00011 0110011

I2 - 0000000 00011 00011 000 00011 0110011

I3 - 0000000 00010 00101 000 00101 0110011

I4 -0000000 00000 00001 000 00111 0110011