

0512.4490 Advanced Computer Architecture Lab 2022  
Lab #1: ASIC hierarchical verification: high level simulator

When designing a new processor, we typically construct simulators of the new processor before designing the architecture details, and before coding it in RTL. This allows us to catch bugs early in the design process, as well as provide a reference for future lower level simulators, and RTL verification.

In this lab we'll model a simple processor by constructing a C high level ISS (instruction set simulator). We'll build the simulator, and test its functionality.

### **SP (simple processor) specification**

SP is a 32 bit processor. It contains 6 registers names r2 to r7, each 32 bits. A single static SRAM of size 65536 words of 32-bit each is used for both program and data. A single 32-bit instruction format is used:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
< 0 > <---opcode---> <-dst--> <-src0-> <-src1-> <-----immediate----->
```

dst specifies the destination register r2 .. r7, or r0 in case no register write back is performed.

src0, src1 specify the two source registers r2 to r7. 0 in the source field will mean 0, and 1 will mean the immediate field, sign extended.

opcode specifies the instruction operation, and is one of the following:

#### Arithmetic/Logic operations:

$R[dst] = R[src0]$  operation  $R[src1]$ , where operation depends on the opcode, and  $R[0]$ ,  $R[1]$  have special meaning as described above:

- 0: ADD: addition:  $R[dst] = R[src0] + R[src1]$
- 1: SUB: subtraction:  $R[dst] = R[src0] - R[src1]$
- 2: LSF: left shift:  $R[dst] = R[src0] \ll R[src1]$
- 3: RSF: (signed) right shift:  $R[dst] = R[src0] \gg R[src1]$
- 4: AND: bitwise logical and:  $R[dst] = R[src0] \& R[src1]$
- 5: OR: bitwise logical or:  $R[dst] = R[src0] | R[src1]$
- 6: XOR: bitwise logical xor:  $R[dst] = R[src0] \wedge R[src1]$
- 7: LHI: load the high bits [31:16] of the destination register with immediate[15:0]

#### Load/Store

- 8: LD: loads memory contents at address specified by  $R[src1]$
- 9: ST: writes  $R[src0]$  to memory at address  $R[src1]$

#### Flow Control

- 16: JLT: jump to immediate[15:0] if  $R[src0] < R[src1]$
- 17: JLE: jump to immediate[15:0] if  $R[src0] \leq R[src1]$
- 18: JEQ: jump to immediate[15:0] if  $R[src0] == R[src1]$
- 19: JNE: jump to immediate[15:0] if  $R[src0] \neq R[src1]$
- 20: JIN: indirect jump to address  $R[src0]$

Each jump instruction saves the address form which it jumped to register r7 (only if the jump was taken).

- 24: HLT: halt execution

Execution starts at PC 0. PC increments by 1 after every instruction except in the case a jump is taken, in which case PC is set to the jump target address. Execution after address 0xffff resumes at address 0.

## **Building a High Level ISA simulator**

You will write a high level ISA simulator for the SP architecture named *iss*. It should accept a text file containing a memory dump of the SRAM, each line containing 8 hexadecimal digits, simulate the program till HALT is encountered, and generate an output memory image as well as trace file as follows:

```
--- instruction 0 (0000) @ PC 0 (0000) -----
pc = 0000, inst = 0088000f, opcode = 0 (ADD), dst = 2, src0 = 1, src1 = 0, immediate = 0000000f
r[0] = 00000000 r[1] = 0000000f r[2] = 00000000 r[3] = 00000000
r[4] = 00000000 r[5] = 00000000 r[6] = 00000000 r[7] = 00000000
```

```
>>>> EXEC: R[2] = 15 ADD 0 <<<<
```

```
--- instruction 1 (0001) @ PC 1 (0001) -----
pc = 0001, inst = 00c80001, opcode = 0 (ADD), dst = 3, src0 = 1, src1 = 0, immediate = 00000001
r[0] = 00000000 r[1] = 00000001 r[2] = 0000000f r[3] = 00000000
r[4] = 00000000 r[5] = 00000000 r[6] = 00000000 r[7] = 00000000
```

```
>>>> EXEC: R[3] = 1 ADD 0 <<<<
```

An example trace is available in `lab1_example.zip`, as well as a simple assembler. The above trace is for the first two commands of the following program:

```
asm_cmd(ADD, 2, 1, 0, 15); // 0: R2 = 15
asm_cmd(ADD, 3, 1, 0, 1); // 1:
asm_cmd(ADD, 4, 1, 0, 8); // 2:
asm_cmd(JEQ, 0, 3, 4, 11); // 3:
asm_cmd(LD, 5, 0, 2, 0); // 4:
asm_cmd(ADD, 2, 2, 1, 1); // 5:
asm_cmd(LD, 6, 0, 2, 0); // 6:
asm_cmd(ADD, 6, 6, 5, 0); // 7:
asm_cmd(ST, 0, 6, 2, 0); // 8:
asm_cmd(ADD, 3, 3, 1, 1); // 9:
asm_cmd(JEQ, 0, 0, 0, 3); // 10:
asm_cmd(HLT, 0, 0, 0, 0); // 11:
for (i = 0; i < 8; i++)
    mem[15+i] = i;
```

The above program is embedded in the `asm.c` assembler. Run the assembler as “`asm example.bin`”, which will generate the following memory dump, that will be an input to the simulator, which will generate the trace:

```
0088000f
00c80001
01080008
241c000b
11420000
00910001
11820000
01b50000
12320000
00d90001
24000003
30000000
00000000
00000000
00000000
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
```

## Assignments

### 1. Warm-up questions:

Answer the following questions in your final report. Whenever you are asked to write a command, specify all of its arguments (dst, src0, src1, imm), but no need to assemble it into a binary string.

1. Write the command which will load the constant 10 into register 5.
2. Write the command to load the constant (-512) into register 5.
3. Write the command to calculate  $R3 - R4$ , and save it into R4.
4. Write a command that will jump to imm only if this imm is greater than R3.
5. Explain how to load a 32 bit constant into a register.
6. Explain how subroutine calls can be implemented on this processor, refer to nested subroutine call handling.

### 2. Example program

Refer to the example assembly program in `asm.c` and answer the following questions in your report:

1. What does it compute?
2. Where are the inputs stored?
3. Where are the outputs stored?
4. Submit a commented version of the assembly program, which contains comments in pseudo code explaining what each line does (the first two lines were commented for you).
5. Suggest how you can rewrite the example program having fewer references to memory

### 3. ISS simulator Testing #1

Write a program which adds two numbers in sign magnitude format. In this format the MSB is the sign bit where 1 indicates a negative number and 0 indicates a positive number. The remaining bits are the magnitude. Remember that if the two numbers have different signs, then this is a subtraction with the result sign being that of the number with the larger magnitude and if the signs are identical then it is addition with the result sign remaining the same. The two input numbers are stored at locations 1000 and 1001. At location 1002 will be the result in sign magnitude format. Overflow bits can be ignored.

Your program must be time efficient. Please demonstrate this issue in your report. Submit:

- a. In your report provide an annotated (explained with comments) assembly code of the program.
- b. Test with the ISS, and submit the following files:
  - `add.bin` - the program machine code, generated by the `asm.c` file
  - `add_trace.txt` - the trace of the execution
  - `add_sram_out.txt` - the memory output image, exhibiting correct data in location 1000-1002

### 4. ISS simulator Testing #2

Write a program which calculate the square root of a number. The output must be only the quotient portion of the number. For example, the square root of 108 is 10.392. The answer desired is 10. Please use a number greater than 20000. The input is placed in location 1000 in memory and the output is in location 1001

Your program must be time efficient. Please demonstrate this issue in your report. Submit:

- a. In your report provide an annotated (explained with comments) assembly code of the program.
- b. Test with the ISS, and submit the following files:
  - `sqrtq.bin` - the program machine code, generated by the `asm.c` file
  - `sqrtq_trace.txt` - the trace of the execution
  - `sqrtq_sram_out.txt` - the memory output image, exhibiting correct data in locations 1000-1001

### 5. ISS simulator implementation

Submit your code for the ISS simulator. It must compile using “make”, producing an executable `iss`, which can be run as:

```
iss code.bin
```

Input: text file with 8 hexadecimal symbols at each line, representing the image of the initial memory state, containing the program code. The file-name of the input is specified by the single command line argument.

Outputs: Trace file should be stored to the file `trace.txt`. The memory output should be outputted to `sram_out.txt`.

Remarks:

- Your ISS simulator will be tested with test programs (you don't get their source code) to verify its correctness.
- Make sure to handle corner cases so that your simulator doesn't crash or exit under any circumstance except reaching the HLT instruction.
- You're free to implement undefined behavior in a way which will be easiest from the hardware point of view. For example – don't terminate the execution of the simulator upon every problem (e.g. bad opcode, bad register), instead try designing a behavior which could be implementable in hardware.
- Both correctness (including corner cases), and performance/efficiency are taken as criteria for the grade.

## 1. Warm-up questions:

Answer the following questions in your final report. Whenever you are asked to write a command, specify all of its arguments (dst, src0, src1, imm), but no need to assemble it into a binary string.

1. Write the command which will load the constant 10 into register 5.
2. Write the command to load the constant (-512) into register 5.
3. Write the command to calculate  $R3 - R4$ , and save it into  $R4$ .
4. Write a command that will jump to imm only if this imm is greater than  $R3$ .
5. Explain how to load a 32 bit constant into a register.
6. Explain how subroutine calls can be implemented on this processor, refer to nested subroutine call handling.

1. LD 0 0 5 10
2. LD 0 0 5 -512
3. SUB 4 3 4 0

4. ?

5. ① ADD 1 0 0 low 16

② LHI 2 0 0 high 16

③ ADD 2 1 2 0

6. ?

## 2. Example program

Refer to the example assembly program in `asm.c` and answer the following questions in your report:

1. What does it compute?
2. Where are the inputs stored?
3. Where are the outputs stored?
4. Submit a commented version of the assembly program, which contains comments in pseudo code explaining what each line does (the first two lines were commented for you).
5. Suggest how you can rewrite the example program having fewer references to memory

$$\begin{cases} R_1 = 0 \\ R_2 = 15 \\ R_3 = 2 \\ R_4 = 8 \end{cases}$$

$R_3 = R_4 ?$  jump to 11

loop id, pile 15

$R_5 = \text{mem}[R_2]$

$R_2 \pm R_1 + 1$

$R_6 = \text{mem}[R_2]$

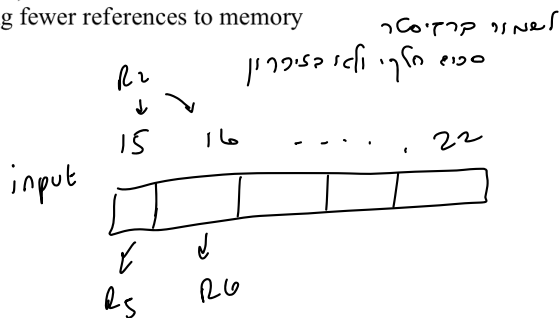
$R_6 = \text{mem}[R_2] + \text{mem}[R_2 + 1]$

$\text{mem}[R_2] = R_6$

$R_3 \pm 1$

$\text{mem}[15-22]$

$\text{mem}[22]$



while  $R_3 \neq R_4$

accumulate sum

### 3. ISS simulator Testing #1

Write a program which adds two numbers in sign magnitude format. In this format the MSB is the sign bit where 1 indicates a negative number and 0 indicates a positive number. The remaining bits are the magnitude. Remember that if the two numbers have different signs, then this is a subtraction with the result sign being that of the number with the larger magnitude and if the signs are identical then it is addition with the result sign remaining the same. The two input numbers are stored at locations 1000 and 1001. At location 1002 will be the result in sign magnitude format. Overflow bits can be ignored.

Your program must be time efficient. Please demonstrate this issue in your report. Submit:

- In your report provide an annotated (explained with comments) assembly code of the program.
- Test with the ISS, and submit the following files:
  - add.bin - the program machine code, generated by the asm.c file
  - add\_trace.txt - the trace of the execution
  - add\_sram\_out.txt - the memory output image, exhibiting correct data in location 1000-1002

+ +

add

0

- -

add

1

+ - / - +

SUB

check greater

+ -

LD 2 0 1000 0 // #0 R2 = mem[1000] = A

LD 3 0 1001 0 // #1 R3 = mem[1001] = B

ADD 4 0 0 1 // # R4 = 1

ADD 5 0 0 31 // # R5 = 31

LSF 4 4 5 0 // #

SUB 5 4 0 1 // #

AND 5 2 4 0 // #

AND 6 3 4 0 // #

AND 2 2 5 0 // #

AND 3 3 5 0 // #

JEQ 0 6 7 // #

R4 = 100...0  
31b

R5 = 0111...1  
31b

R6 = MSB A = sign A (10...0)  
(00...0)

R7 = MSB B = sign B

R2 = magnitude A

R3 = magnitude B

jump to <sup>ADD</sup> — if R6 = R7

ADD	S	2	3	0	// #	R <sub>S</sub> add magnitude	}	ADD
OR	S	5	6	0	// #	R <sub>S</sub> = sign   R <sub>S</sub> none		
JEQ	0	0	0	-	// #	jumps to finish		

SUB	S	2	3	0	// #	R <sub>S</sub> sub magnitude	}	SUB
JLT	0	2	3	-	// #	jump if mag A < mag B		
OR	S	5	6	0	// #	R <sub>S</sub> = sign <sub>A</sub>   R <sub>S</sub>		
JEQ	0	0	0	-	// #	jump to finish		
OR	S	5	7	0	// #	R <sub>S</sub> = sign <sub>B</sub>   R <sub>S</sub>		

ST	0	S	1002	0	// #	store result	}	finish
HLT	0	0	0	0	// #	halt		