0512.4490 Advanced Computer Architecture Lab 2021
Lab #2: ASIC hierarchical verification: low level simulator

In the previous lab, we designed a high level instruction set simulator for SP processor. In this lab, we'll build a low level cycle accurate simulator for the processor, and verify its behavior against the high level ISS.

In contrast to the high level ISS which focuses only on the programmer visible instruction set architecture (ISA), the low level simulator describes a specific micro architecture implementation. There can be many different micro architecture implementations of the same processor ISA, each one with different performance and power characteristics, but all compatible from the instruction set point of view (e.g. pipelined and non-pipelined).

The low level simulator, in contrast to the high level simulator, also keeps track of the exact timing of each command, and can tell us the state of each register, each cycle.

SP micro architecture

We'll design a low performance, serial (non pipelined) implementation of the processor, consisting of the following micro architecture:

Registers

r2 – r7: [31:0]: 6 32 bit registers, holding the ISA general purpose registers.
pc: [15:0] 16 bit program counter.
inst: [31:0] 32 bit instruction.
opcode [4:0]: 5 bit opcode.
dst[2:0]: 3 bit destination register index.
src0[2:0]: 3 bit source #0 register index.
src1[2:0]: 3 bit source #1 register index.
immediate[31:0]: sign extended immediate.
alu0[31:0]: 32 bit alu operand #0.
alu1[31:0]: 32 bit alu operand #1.
aluout[31:0]: 32 bit alu output.
cycle_counter[31:0]: 32 bit cycle counter.
ctl_state[2:0]: control logic state machine register.

Memory

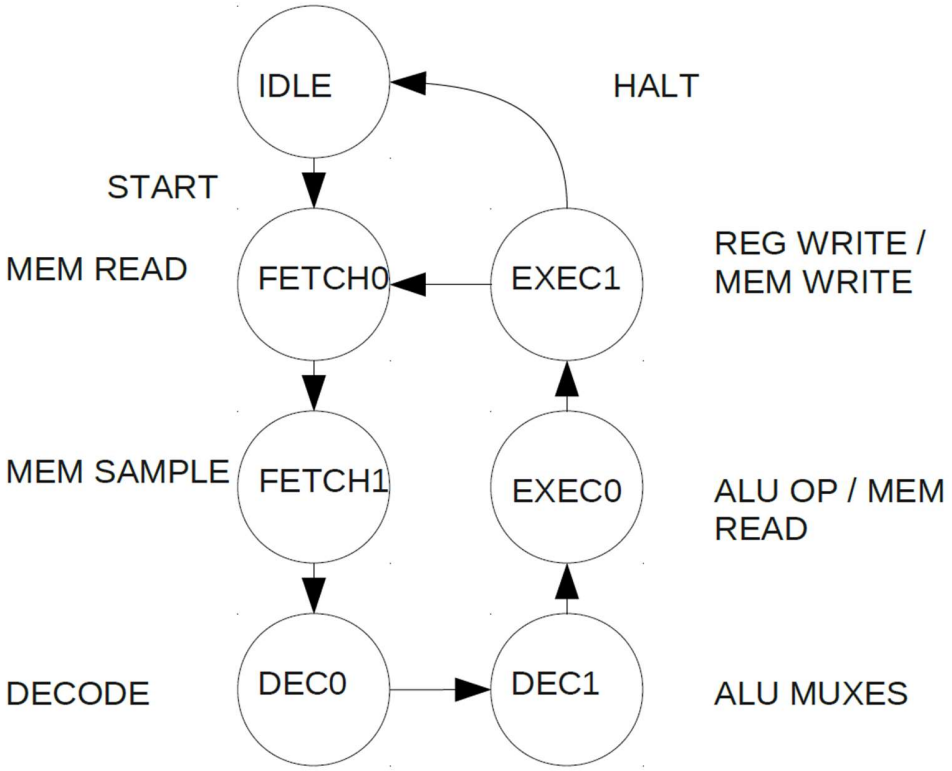
Single synchronous SRAM, 65536 lines x 32 bit each. Each cycle either a read or a write can be performed. A read operation will return the data next cycle.

ALU

The ALU operates on two 32-bit operands alu0[31:0] and alu1[31:0], performs the operation specified by the opcode, and stores the output in aluout[31:0]. For ADD,SUB,LSF,RSF,AND,OR,XOR, it performs the requested operation. For LHI, it performs (alu1 << 16) | alu0[15:0]. For JLT, JLE, JEQ, JNE, it compares ALU0 and ALU1, returning 1 in aluout if the compare succeeded, and 0 otherwise. For the rest of the commands, the ALU doesn't do anything.

Control logic

The control logic will consist of a state machine with 7 states, each taking one cycle:



IDLE (state #0): Waits till start signal is given before processing to FETCH0.
FETCH0 (state #1): Issues read command to memory to fetch the current instruction from address PC.
FETCH1 (state #2): Samples memory output to the inst register.
DEC0 (state #3): Decodes the instruction register into its fields: opcode, dst, src0, src1, and immediate (sign extended).
DEC1 (state #4): Prepares the ALU operands.
EXEC0:(state #5): Executes ALU and LD operations.
EXEC1: (state #6): Write backs ALU and memory (ST). On HLT goes to IDLE, otherwise loops back to FETCH0.

Building a Low Level ISA simulator

Low level simulation engine

In contrast to the high level simulator, the low level simulator simulates processor cycles rather than instructions. You'll build a processor model within a pre-built simulation engine. The simulator engine is in file llsim.c, and keeps tracks of the registers and memories within the design. Your processor design should be completed in the file sp.c, which is partially written. Each cycle, the sp_run() function is called to simulate the execution of a single processor cycle.

Simulating registers

Each register is simulated with two copies, one containing its "old" value at the current cycle, corresponding to the 'Q' output of the D flip flop, and the other copy containing its "new" value at the next cycle, corresponding to the 'D' input of the flip flop.

Your "run" function should contain a sequence of instructions setting "new" values of the registers as function of the "old" values. The "old" values should never be written to, only read from, and the "new" values should never be read from, only written to. At the end of the simulation cycle, the simulation engine will copy the "new" values to the "old" values, simulating the DFF behavior.

The old registers are accessed using "spro->", (sp registers old), and the new registers are accessed using "sprn->" (sp registers new). For example, the following command increments a counter every cycle:

```
sprn->cycle_counter = spro->cycle_counter + 1;
```

Note that the current value of cycle_counter is not yet modified till the next cycle, so if, say, it is 3, spro->cycle_counter remains 3 for the entire execution of your run () function.

All registers have been defined in the sp_registers_t structure.

Simulating Memory

The simulator engine provides the following functions for memory access:

```
void llsim_mem_read(llsim_memory_t *memory, int addr);
int llsim_mem_extract_dataout(llsim_memory_t *memory, int msb, int lsb);
void llsim_mem_set_datain(llsim_memory_t *memory, int val, int msb, int lsb);
void llsim_mem_write(llsim_memory_t *memory, int addr);
```

A read is performed by calling llsim_mem_read with the memory pointer and the address. Data will only be returned next cycle, and can be extracted using the llsim_mem_extract_dataout() function. msb and lsb specify partial bits (most significant and least significant) from the data.

A write is performed by calling llsim_mem_set_datain() to set the write data, and then, in the same cycle, calling llsim_mem_write().

There are additional inject/extract functions used to initialize and extract the memory contents, but those shouldn't be used during the run() functions, only at the initialization, and after the simulation ends, to extract the results for debugging.

Assignments

Question 1: Understanding registers simulation

1. Suppose at the current cycle, r[2] = 12, r[3] = 45, r[4] = 9, and r[5] = 22, and the following commands are written in the run() function:
   ```
   sprn->r[4] = spro->r[2] + spro->r[3];
   sprn->r[5] = spro->r[3] - spro->r[4];
   ```
   After execution of 2 cycles, what will be the values of r2, r3, r4, and r5?

2. The following code is wrong, why?
   ```
   spro->r[1] = spro->r[2] + sprn->r[3];
   ```

Question 2: Understanding micro-architecture

1. How many clock cycles will it take to execute *n* instructions on the given micro-architecture?
2. Can a proposed *microarchitecture* support memory access every clock cycle? (for example to read different word every cycle) ?
3. What are the advantages and the disadvantages of the presented micro-architecture, relative to a pipelined one?

Question 3: Low level simulator

Write a low level simulator for the SP architecture by filling the `sp_ctl()` function in the `sp.c` template provided to you. The input to the simulator is the same example.bin file as the input used for the high-level simulator in lab #1. Run the simulator using `./llsim example.bin`. The simulator should generate a memory dump file two trace files and:

1. `sram_out.txt` - depicts the memory contents upon the simulation completion.
2. `inst_trace.txt` - an instruction execution trace exactly equal to the trace generated by the high level simulator you wrote in lab #1, so they can be binary compared.
3. `cycle_trace.txt` - a cycle execution trace containing a dump of all registers every cycle, which begins with the cycle number, and then contains a list of all registers in hex, one in each line.

Example of an instruction trace:

```
--- instruction 21 (0015) @ PC 5 (0005) -----------------------------------------------------------
pc = 0005, inst = 00910001, opcode = 0 (ADD), dst = 2, src0 = 2, src1 = 1, immediate = 00000001
r[0] = 00000000 r[1] = 00000001 r[2] = 00000013 r[3] = 0000000a
r[4] = 00000004 r[5] = 00000014 r[6] = 00000000 r[7] = 00000006

>>>> EXEC: R[2] = 19 ADD 1 <<<<
```

Example of a cycle trace:

```
cycle 33
r2 0000000f
r3 00000000
r4 00000000
r5 00000014
r6 00000000
r7 00000000
pc 00000005
inst 00910001
dst 00000003
src0 00000003
src1 00000004
immediate 00000000
alu0 00000000
alu1 00000000
aluout 00000000
```

cycle_counter 00000021
ctl_state 00000003

When you encounter HLT, dump_sram() is called to dump the sram to sram_out.txt, llsim_stop() to stop the simulation at the next cycle.

Example files example.bin (same as lab #1), `inst_trace.txt`, `cycle_trace.txt`, and `sram_out.txt` are provided in Moodle.

Submit your `sp.c` file.

Question 4. low level simulator Testing #1

Verify the same `mult.bin` file you used to test the high level simulator in lab #1. As a reminder, it will multiply two (possibly signed) numbers. The two input numbers are stored at addresses 1000 and 1001, and the result will be written to 1002. Test the low level simulator with this code, and submit the `mult_inst_trace.txt`, `mult_cycle_trace.txt`, and `mult_sram_out.txt`.

Question 5. low level simulator Testing #2

Verify the same `fibo.bin` file you used in lab #1. As a reminder, it will calculate the first 100 elements of the Fibonacci sequence, and write it to memory starting at address 1000. Test the low level simulator with this code, and submit the `fibo_inst_trace.txt`, `fibo_cycle_trace.txt`, and `fibo_sram_out.txt`.

Question 6. Background DMA (direct memory access) copy

Design and add to the processor a DMA state machine capable of copying a block of memory in the background, in parallel to continuing execution of the main program assembly code:

- The background copy is started using a new opcode. It should accept as parameters the memory source address, destination address, and length. Once started, it'll copy the data word by word in ascending direction.
- Another new opcode will be used by the program to poll the copy status.

Submit the following:

1. In the report describe your implementation of the DMA and of the two new instructions it required.
2. Submit the `sp.c` file with the addition of the copy state machine.

Question 7. DMA testing

Write a test program for the DMA copy machine. The program should perform a copy using the machine, and test that the copy was done correctly using assembly instructions. The test result should be in r2 (0 fail, 1 pass). Make sure to stress the state machine by performing memory access from the program assembly code in parallel to the machine background operation. Require (and test) the ability of the DMA copy to cope with overlapping source and delay sections:

e.g. DMA 50 60 50 --> will overwrite the 60... addresses before the original data was read from them.

Submit the following:

1. In your report provide an annotated (explained with comments) assembly code of the program.
2. Test with the ISS, and submit the following files:
    - `dma.bin` - the program machine code, generated by the asm.c file
    - `dma_inst_trace.txt` – the trace of the execution
    - `dma_cycle_trace.txt` – the trace of the execution
    - `dma_sram_out.txt` – the memory output image