3 Stage Pipeline Processor

Term Project Part-2 Report

Group 9:
CS14B023 Rahul Kejriwal
CS13B005 Bharat Sai Botta
CS16S033 Debanjan Ghatak

Table of Contents:

Assumptions & Framework:

1. We have written the SVA in the testbench itself and simulated using the Synopsys VCS 2014.10 Simulator on the EDA Playground platform.

Module-wise Report:

1. Fetch Unit

a. Properties Written:

i. Assert resetting of program counter (PC) and output instruction of Fetch unit on resets:

```
property P1;
       @(posedge rst) (instr_addr == 8'b0 and curr_instr  == 16'b0);
endproperty
```

ii. Assert incrementing of PC every cycle:

```
property P2;
       reg[7:0] prev_progctr;

       @(posedge clk) (1,prev_progctr = instr_addr)
              ##1 (instr_addr == ((prev_progctr + 1)&8'b11111111));
endproperty
```

iii. Assert forwarding of instruction fetched from memory every cycle:

```
property P3;
       reg[15:0] old_instr;

       @(posedge clk) (1, old_instr = instr)
              ##1 curr_instr == old_instr;
endproperty
```

iv. Assert looping back of address to 0 after 255:

```
property P4;
       @(posedge clk) instr_addr == 8'b11111111 |->
              ##1 instr_addr == 8'b0;
endproperty
```

b. Errors Debugged:

i. No bugs found during property verification.

c. Completeness & Consistency Reasoning:

i. The fetch unit needs to fetch the instruction at address PC and increment the PC each cycle. This is asserted by P2 and P3.
ii. After reaching the end of the address space, i.e., 255, the PC should loop back to 0 (This is the semantics we fixed and assumed at the start of the project). This is asserted by P4.
iii. Also, we assert zeroing of PC and output instruction at reset using P1. (In our project semantics, instruction 0x0 corresponds to a NOP and has no effect on the program context.)

2. Decode & Fetch Operand Unit

a. Properties Written:

i. Assert forwarding of decoded opcode each cycle:

```
property P1;
       @(posedge clk) (!$rose(rst)) |-> opcode == instr[15:12
endproperty
```

ii. Assert decoding of non-LOAD/STORE instructions to obtain source registers and sending of source register addresses to Register File and Execute Unit as necessary:

```
property P2;
       @(posedge clk) (!$rose(rst) and instr[15:12] != 4'b1110 and
              instr[15:12] != 4'b1111) |->
              (nextDestReg == instr[11:8] and destReg == instr[11:8]
              and srcReg1 == instr[7:4] and srcReg2 == instr[3:0]);
endproperty
```

iii. Assert forwarding of decoded memory address and destination register for LOAD instruction:

```
property P3;
       @(posedge clk) (!$rose(rst) and instr[15:12] == 4'b1110) |->
              (nextDestReg == instr[ 3:0] and destReg == instr[ 3:0]
              and memAddr == instr[11:4]);
endproperty
```

iv. Assert decoding of STORE instruction to obtain memory address and source register and forwarding of memory address to Execute Unit and source register to Register File:

```
property P4;
       @(posedge clk)  (!$rose(rst)) |->
              (instr[15:12] == 4'b1111) |->
              (srcReg1 == instr[ 3:0] and memAddr == instr[11:4]);
endproperty
```

v. Assert forwarding of data requested from Register File:

```
property P5;
       @(posedge clk) (!$rose(rst)) |->
              (srcVal1 == srcRegVal1 and srcVal2 == srcRegVal2
              and used1 == inuse1 and used2 == inuse2);
endproperty
```

vi. Assert reset properties:

```
property P6;
       @(posedge clk) $rose(rst) |-> (srcReg1 == 4'b0
              and srcReg2 == 4'b0
              and nextDestReg == 4'b0
              and opcode  == 4'b0
              and destReg == 4'b0
              and srcVal1 == 16'b0
              and srcVal2 == 16'b0
              and memAddr == 8'b0
              and used1   == 0
              and used2   == 0);
endproperty
```

b. Errors Debugged:

i. Found a bug with reset signal timing:

The semantics of the processor is that the cycle in which reset signal rises is wasted and execution begins again from the next cycle.

But in our earlier code, in the cycle where reset was asserted, first the processor reset-ed to initial condition and in the same cycle executed a new instruction. This would interfere with the timing of other modules.

We fixed this bug by resolving the timing issue.

c. Completeness & Consistency Reasoning:

i. The Decode and Fetch unit is responsible for decoding the message into relevant opcode, source registers, destination register and/or memory address. It has 3 different modes of operation based on different types of instruction: non-LOAD/STORE instructions, LOAD instructions and STORE instructions.

ii. For all modes of operation, this unit is responsible for correctly forwarding the opcode of the instruction. This is asserted by P1.

iii. For non-LOAD/STORE instructions, it has to decode from the instruction, 2 source registers and a destination register. It then has to lookup the source register values and their inuse status from the register file. It should also instruct the Register File to make the destination register inuse for the next cycle. This is asserted by P2. It is also responsible for forwarding the looked up information from register file to the Execute unit and this is asserted by P5.

iv. For LOAD instruction, it has to decode the memory address and destination register from the instruction and forward them to the Execute unit. It should also instruct the Register File to make the destination register inuse in the next cycle. This is asserted by P3.

v. For STORE instructions, it has to decode the memory address and source register from instruction and forward the memory address to the Execute unit and lookup the source register value and inuse status from the Register File. This is asserted by P4. It is also responsible to forward the lookup up information from the Register File to the Execute Unit. This is asserted by P5.

vi. We also assert returning to initial state on assertion of reset signal. This is done by P6.

3. Register File Unit

a. Properties Written:

i. Assert lookup of source register values and inuse status when requested by Decode & Fetch Operand unit:

```
property P1;
       @(posedge clk)
              srcRegVal1 == R.r[srcReg1] and
              srcRegVal2 == R.r[srcReg2] and
              inuse1     == R.inuse[srcReg1] and
              inuse2    == R.inuse[srcReg2];
endproperty
```

ii. Assert setting of inuse bit for new destination register and resetting of inuse bit of old destination register for next cycle:

```
property P2;
       reg[7:0] next_dest;
       reg[7:0] dest;

       @(posedge clk) (1, next_dest=nextDestReg, dest=destReg)
              ##1 ((dest == next_dest or R.inuse[dest] == 0) and
                   (R.inuse[next_dest] == 1));
endproperty
```

iii. Assert writing of value to register when requested by Execute unit:

```
property P4;
       @(posedge clk) $rose(storeDone) |-> R.r[destReg] == destVal;
endproperty
```

iv. Assert reset of outputs:

```
property P5;
        @(posedge clk) $rose(rst) |->
                srcRegVal1 == 16'b0
                and srcRegVal2 == 16'b0
                and inuse1 == 0
                and inuse2 == 0
                and storeDone == 0;
endproperty
```

v. Assert reset of register values and inuse bits:

```
property ResetProp(i);
        @(posedge clk) $rose(rst) |-> R.r[i] == 0 and R.inuse[i] == 0;
endproperty
```

vi. Assert write acknowledgement eventually for a write request:

```
property P7;
        @(posedge storeNow) ##[1:$] $rose(storeDone);
endproperty
```

b. Errors Debugged:
   i. Found a bug w.r.t. Lookup of values for Decode & Fetch Operand Unit:

      The semantics of the Decode & Fetch Operand Unit is that it sends address of source and destination register to this unit for looking up value and inuse status of source registers and to assert inuse bit of destination register in next cycle.

      When we have 2 instructions one after the other such that both have the same source and destination registers (ex: a = a+b followed by a = a+b, where a and b are 2 registers), the earlier version of the code did not respond thinking that the Decode & Fetch Operand unit didn't actually make any request yet. However, the values of these registers might have changed from the last cycle.

      We fixed this bug by refreshing the looked up data at the beginning of each cycle even if input addresses from Decode & Fetch Operand Unit didn't change in the cycle.

c. Completeness & Consistency Reasoning:
   i. This unit interacts with 2 other units: Decode & Fetch Operand Unit and the Execute Unit. Thus, it has two responsibilities.
   ii. Firstly, w.r.t. the Decode & Fetch Operand Unit, it has to lookup the register values and inuse status bits for source registers given as input

and has to assert the inuse bit of the destination register for the next cycle. This is asserted by P1 and P2.

iii. Secondly, w.r.t. The Execute Unit, it has to write the destination value to the destination register when it receives a write request from Execute Unit. It should send back a write acknowledgement to the Execute Unit after it finishes writing the value. Additionally, it must eventually send a write acknowledgement whenever a write request is made to ensure the write actually took place. This is asserted by P4 and P7.
Also, the inuse bit of the register to which write took place should also be deasserted (if it is not the destination register for next cycle also). This is asserted by P2.

iv. We also assert returning to initial state on reset via P5 and ResetProp. On reset, the output from the unit are set to 0 along with all the register values and inuse bits.

4. Execute Unit

a. Properties Written:

i. Assert property for addition without overflow:

```
property P2;
       @(posedge clk) (opcode == 4'b0010) |->
              ##0 ({ProcessorStatusWord[15],destVal} ==
              executeAndStoreBack.val1 + executeAndStoreBack.val2);
endproperty
```

ii. Assert property for addition with overflow:

```
property P3;
       @(posedge clk) (opcode == 4'b0010 and
       executeAndStoreBack.val1[15] == executeAndStoreBack.val2[15]
       and destVal[15] != executeAndStoreBack.val1[15]) |->
              ##0 ({ProcessorStatusWord[15],destVal} ==
              executeAndStoreBack.val1 + executeAndStoreBack.val2
              and ProcessorStatusWord[14] == 1);
endproperty
```

iii. Assert property for substraction without overflow:

```
property P4;
       @(posedge clk) (opcode == 4'b0011) |->
              ##0 ({ProcessorStatusWord[15],destVal} ==
              executeAndStoreBack.val1 - executeAndStoreBack.val2);
endproperty
```

iv. Assert property for substraction with overflow:

```
property P5;
        @(posedge clk) (opcode == 4'b0011 and
        executeAndStoreBack.val1[15] != executeAndStoreBack.val2[15]
        and destVal[15] != executeAndStoreBack.val1[15]) |->
                ##0 ({ProcessorStatusWord[15],destVal} ==
                executeAndStoreBack.val1 - executeAndStoreBack.val2
                and ProcessorStatusWord[14] == 1);
endproperty
```

v. Assert property for multiplication:

```
property P6;
        @(posedge clk) (opcode == 4'b0100) |->
                ##0 ({ProcessorStatusWord[15],destVal} ==
                executeAndStoreBack.val1 * executeAndStoreBack.val2);
endproperty
```

vi. Assert property for shift left operation:

```
property P7;
        @(posedge clk) (opcode == 4'b0101) |->
                ##0 ({ProcessorStatusWord[15], destVal} ==
                executeAndStoreBack.val1 << executeAndStoreBack.val2);
endproperty
```

vii. Assert property for shift right operation:

```
property P8;
        @(posedge clk) (opcode == 4'b0110) |->
                ##0 ({ProcessorStatusWord[15],destVal} ==
                executeAndStoreBack.val1 >> executeAndStoreBack.val2);
endproperty
```

viii. Assert property for bitwise and operation:

```
property P9;
        @(posedge clk) (opcode == 4'b0111) |->
                ##0 (destVal== (executeAndStoreBack.val1 &
                executeAndStoreBack.val2));
endproperty
```

ix. Assert property for bitwise or operation:

```
property P10;
        @(posedge clk) (opcode == 4'b1000) |->
                ##0 (destVal== (executeAndStoreBack.val1 |
                executeAndStoreBack.val2));
endproperty
```

x. Assert property for negation operation:

```
property P11;
        @(posedge clk) (opcode == 4'b1001) |->
                ##0 (destVal== ~(executeAndStoreBack.val1));
endproperty
```

xi. Assert property for bitwise xor operation:

```
property P12;
        @(posedge clk) (opcode == 4'b1010) |->
                ##0 ( destVal== (executeAndStoreBack.val1 ^
                executeAndStoreBack.val2));
endproperty
```

xii. Assert property for load operation:

```
property P13;
        @(posedge clk) (opcode == 4'b1110) |->
                ##0 (memAddrLoadStore == memAddr and readReq == 1'b1);
endproperty
```

xiii. Assert property for store operation:

```
property P14;
        @(posedge clk) (opcode == 4'b1111) |->
                ##0 (memAddrLoadStore == memAddr and memValueStore ==
                executeAndStoreBack.val1 and writeReq == 1);
endproperty
```

xiv. Assert property for checking all the synchronisation signals as well as the last computed value for non-loadstore instructions:

```
property P15;
        @(posedge clk) (opcode != 4'b0000 and opcode != 4'b0001 and
        opcode != 4'b1110 and opcode != 4'b1111 ) |->
                ##0 (destRegStore == destReg and storeNow == 1
                and executeAndStoreBack.LastComputedValue == destVal);
endproperty
```

xv. Assert property for the zero flag bit of the PSW:

```
property P16;
        @(posedge clk) (opcode != 4'b0000 and opcode != 4'b0001 and
        opcode != 4'b1110 and opcode != 4'b1111 and
        {ProcessorStatusWord[15], destVal} == 0) |->
                ##0 (ProcessorStatusWord[13] == 1);
endproperty
```

xvi. Assert property for checking synchronisation signals for loadstore instructions:

```
property P17;
        @(posedge valueReady)
        ##1 (destVal == memValueLoad and readReq == 1 and destRegStore
        == destReg and storeNow == 1 and
        executeAndStoreBack.LastComputedValue == destVal);

endproperty
```

xvii. Assert property for setting the zero flag bit:

```
property P18;
        @(posedge valueReady) (destVal == 0) |->
                ##0 (ProcessorStatusWord[13] == 1);
endproperty
```

b. Errors Debugged:
   i. No bugs were found during property verification.

c. Completeness & Consistency Reasoning:
   i. The execution and storeback unit computes the different arithmetic and logical operations and we have written assertions for all those properties.We have checked whether the result is being computed correctly or not. We have also checked whether the overflow flag bit is getting correctly computed or not.
   ii. We have also verified the properties for the load_store instructions, whether the destination register address is rightly getting copied into destRegStore and the synchronisation signals correctly being asserted or not and all the status flag bits correctly getting computed or not.
   iii. We also checked whether the lastcomputedvalue is correctly getting computed or not when the inuse bits are set to 1.