Course: CA

exercise: 2

date: 29/11/2018

author: Minghao Cheng, 2359434

author: Hao Song, 2358944

author: Boyko Radulov, 2130056

Part 1:

The motivation is to implement a new RX instruction called “loadxi”. The behaviour of “loadxi” instruction can be described briefly using the following example:

loadxi R1,$12ab[R2]

performs R1 := mem[12ab+R2], R2:= R2+1

The “loadxi” instruction can be broken down into two parts (using the previous example):

First, perform R1 := mem[12ab+R2];

Second, perform R2 :=R2+1;

The first part is exactly the same with the “load” instruction, thus, it is doable to use the control algorithm of the “load” instruction to achieve it. However, for the second part, the present datapath is unable to perform it. This is because that the destination address of the register file cannot be set to “ir_sa” (it can only be “ir_d”). To implement “R2:=R2+1”, a new word multiplexer and a new control signal must be introduced.

In the exercise, a new signal called “rf_d” was introduced, which is the destination address of the register file. The signal “rf_d” can either be “ir_sa” or “ir_d” by control the word multiplexer connected to them. The control signal was named “ctl_d_ira”. When the “ctl_d_ira” is set to one, the register file

would take “ir_sa” as its destination address, and when it is set to zero, the destination address would be “ir_d” The code is shown as follow:

```
      (a,b) = regfile n k (ctl_rf_ld ctlsigs) rf_d rf_sa rf_sb p

      rf_d = mux1w (ctl_d_ira ctlsigs) ir_d ir_sa
```

Combining the idea of part1 and part2, the “loadxi” instruction was successfully Implemented. It takes four clock cycles, the first three cycles perform “load” instruction, and the last cycle increments the register which’s address is “ir_sa”. The control algorithm is shown as follow:

```
        7 -> -- loadxi instruction
            st_loadxi0:
              ad := mem[pc], pc++;
                assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc,
                        ctl_alu_abcd=1100, ctl_pc_ld]
            st_loadxi1:
              ad := reg[ir_sa] + ad
                assert [set ctl_y_ad, ctl_alu_abcd=0000,
                        set ctl_ad_ld, ctl_ad_alu]
            st_loadxi2:
              reg[ir_d] := mem[ad]
                assert [ctl_rf_ld]
            st_loadxi3:
              reg[ir_sa] = reg[ir_sa] + 1
                assert [ctl_rf_ld, ctl_alu_abcd = 1100,
                        ctl_rf_alu, ctl_d_ira]
```

Furthermore, the simulation driver “M1run” was also manipulated for correctly showing the register updates. Besides adding new states and control signals, the output part was changed. Originally, it would only take “field ir 4 4” as the index of the register which is updated. When the second part of the “loadxi” instruction is performed, it cannot give the correct index of the register, so a word multiplexer is used to show the correct index.

```
-- Process a load to the register file
         fmtIf (ctl_rf_ld ctlsigs)
           [string "Register file update: ",
            string "R",
            bindec 1 (mux1w (ctl_d_ira ctlsigs) (field ir 4 4) (field
ir 8 4) ),
            string " := ", hex p,
            setStateWs setRfLoad [(mux1w (ctl_d_ira ctlsigs) (field ir
4 4) (field ir 8 4) ), p],
            string "\n"
            ]
```

```
            [],
```

The “loadxi” instruction was tested using the manipulated “arraymax” program. The result is the same as what was expected, and it is correct. It shows that the “loadxi” instruction is suitable for loop control where the loop would load an array at each time. Comparing with the old “arraymax” program which does not use “loadxi” instruction. The new version with “loadxi” instruction is much more efficient. From the perspective of programmer, the old version takes 22 lines of code which the new version takes 20. From the point of speed, the old one takes 193 clock cycles and the new one only takes 177 clock cycles.

Part 2:

This question is about implementing the multiplication instruction in which the clock cycles required by the instruction depends on inputs.

The multiplication circuit takes “x”, ”y” and “start” as its input (k is the word size, it is fixed to 16 in Sigma16 architecture, and the output would be 16 bits as well in this case), and it outputs “prod” and “ready”. When the “start” signal is set to one, the multiplier will perform the calculation and “ready” would be zero, and when the “prod” is valid, the “ready” signal will be one.

The implementation of the “mul” instruction was split into several steps:

First, add the multiplier circuit into the datapath, connect “x” and “y” to the output “a” and “b” of the register file;

Second, add a new control signal called “ctl_mul_start” to trigger the multiplier to start the calculation;

Third, add another signal called “ctl_r_prod” to control the data input of the register file, when “ctl_r_prod” is set to one, the register file will take the output of the multiplier as its data input;

Fourth, add the “ready” bit of the multiplier to the outputs of the datapath, and add a new input bit of the control circuit to receive the bit;

Last, add the states in the control circuit. (In question 1, this part was not mentioned because it was simple, and the implementation was just like the other states. However, for this question, because the clock cycles required by this instruction is not fixed, the implementation is different). The code is shown as follow:

```
    st_mul0   = dff (pRRR!!2)
    st_mul1   = reg1 ready st_mul0
    st_mul2   = reg1 ready st_mul1
```

To describe the implementation of the states, it is needed to state the control algorithm. For “st_mul0”, it would be one if the instruction in the instruction register is actually “mul”. In this state, the control circuit would set “ctl_mul_start” to one to let the multiplier start calculation. In the “st_mul1” state, the CPU is waiting the multiplier to finish the calculation. The register which store the value of “st_mul1” would firstly load a one when “st_mul0” is one, because at state “st_mul0”, “ready” is one. During the calculation, the “ready” signal is zero, so it would not get a new value. When the calculation is finished, “ready” becomes one and “st_mul1” would load a zero. Besides, “st_mul2” would load a one. At state “st_mul2”, the control circuit would set “ctl_r_prod” to one and the register file would load the “prod” into the destination register.

The control algorithm is shown as follow:

```
  2 -> -- mul instruction
       -- st_mul0:
         reg[ir_d] := reg[ir_sa] * reg[ir_sb]
         assert [ctl_mul_start]
         st_mul1:
         # waiting for ready
         st_mul2:
         assert [ctl_rf_ld, ctl_rf_alu, ctl_r_prod]
```

In short words, this implementation of the “mul” instruction is actually “bypassing” the ALU when instruction is performed.

The “mul” instruction was tested by performing two “mul” instructions:

```
mulRun =
 [
  "f101", "000b", -- 0000 start load R1,x1[R0]
  "f201", "000c", -- 0002       load R2,y1[R0]
  "2312",         -- 0004       mul R3,R1,R2
  "f101", "000d", -- 0005 start load R1,x2[R0]
  "f201", "000e", -- 0006       load R2,y2[R0]
  "2312",         -- 0007       mul R3,R1,R2
  "d000",         -- 000a        trap  R0,R0,R0     ; terminate

                  -- Data area
  "0002",         -- 000b x1
  "0003",         -- 000c y1
  "0025",         -- 000d x2
  "0025"          -- 000e y2
 ]
```

The result is correct, and it shows that the first multiplication takes 8 cycles and the second one takes 12 cycles (both of them contains fetch and dispatch). During the calculation, “st_mul1” is always one.