Laboratory Exercise 11

A More Enhanced Processor

In Lab Exercise 10 you made enhancements to the processor from Lab 9, by including a program counter, memory interface, and the `ld`, `st`, `and`, and `b{cond}` instructions. This exercise involves further extensions to the processor design. The numbering of figures and tables in this document are continued from those in Parts I to VII of Lab Exercises 9 and 10.

For this exercise you will augment the processor architecture so that it supports subroutines and stacks, and also provides shift and rotate operations. All of the processor registers will be the same as in Lab 10, except that register $r5$ will be changed into an *up/down counter*, as illustrated in Figure 24. This figure shows only the processor registers $r4, \ldots, r7$ (*pc*) and their connections to the *Buswires* multiplexer; refer to Lab 10 to see a more complete schematic of the processor.

In assembly-language code register $r5$ can be referred to as the *stack pointer* register, *sp*. It is used as an *address* for pushing and popping data on the stack. Since it is an up/down counter, the *sp* can easily be *decremented* before a register is *pushed* onto the stack, and *incremented* after a register has been *popped* off of the stack. The processor's control unit decrements *sp* by using the *sp_decr* signal shown in Figure 24, and increments this register by using the *sp_incr* signal. These signals are just the *up*/*down* control inputs for the counter. Arbitrary data can also be loaded into register $r5$ (*sp*) in the same way as in Lab 10, by using the $r5_{in}$ signal.

The processor will have eight new instructions, which are listed in Table 5. The `push rX` instruction is used to store the contents of a register, *rX*, onto the stack. This instruction first decrements the *sp* (register $r5$), and then stores *rX* into memory at the address in *sp*. The `pop rX` instruction is used to load data into a register *rX* from memory at the address in *sp*. After loading this data, *sp* is then incremented.

The branch instruction, `b{cond}`, was introduced in Lab 10. This exercise defines a new type of branch instruction, `bl Label`, which is used for *subroutine linkage*. This *branch with link* instruction first copies the address of the program counter (which will already have been incremented to point to the *next* instruction after the `bl`), into register $r6$. Then, the `bl` instruction sets the program counter to the address of the subroutine, `Label`. In assembly-language code register $r6$ can be referred to as the *link register*, *lr*. To effect a *return*, a subroutine can use the instruction `mv pc, lr`.

Operation	Function performed
push rX	*sp* ← *sp* −1, [*sp*] ← *rX*
pop rX	*rX* ← [*sp*], *sp* ← *sp* +1
bl Label	$r6$ ← *pc*, *pc* ← *Label*
cmp rX, Op2	performs *rX* − *Op2*, sets flags
lsl rX, Op2	*rX* ← *rX* << *Op2*
lsr rX, Op2	*rX* ← *rX* >> *Op2*
asr rX, Op2	*rX* ← *rX* >>> *Op2*
ror rX, Op2	*rX* ← *rX* <<>> *Op2*

Table 5: New instructions.

The `cmp` instruction is similar to the `sub` instruction that was introduced in Lab 9. This instruction performs the operation *rX* − *Op2*, but only affects the flags. The `cmp` instruction does not modify register *rX*.

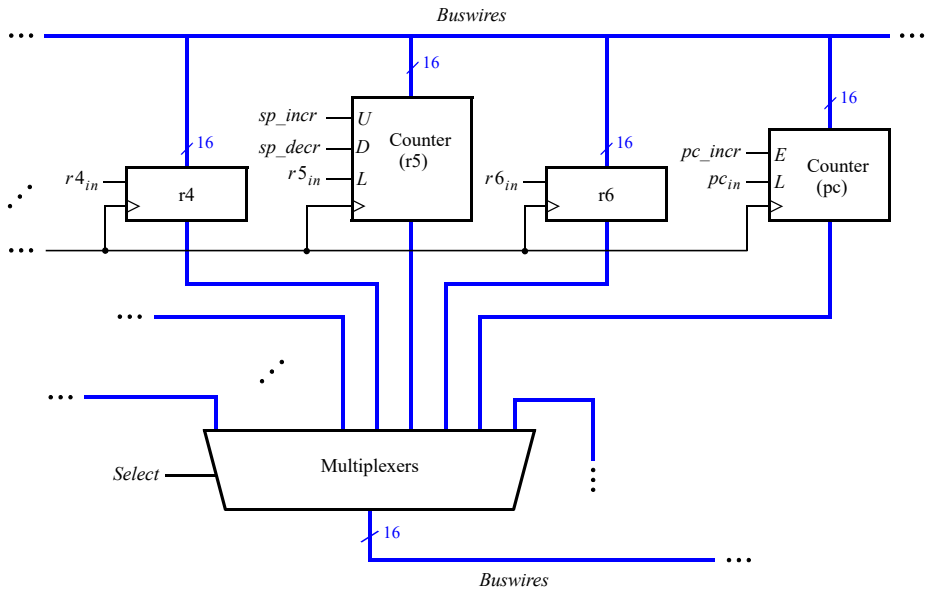

Figure 24: The stack pointer register.

Finally, the `lsl`, `lsr`, `asr`, and `ror` instructions extend the ALU in the processor to provide *shift* and *rotate* capability. The `lsl` instruction performs a logical-shift-left operation (<<). It shifts the contents of register *rX* to the left by the amount specified in *Op2*. The effect of this instruction is to perform *multiplication* by *powers of two*. The maximum possible shift amount is 15. It can be given in the form of immediate data, *#D*, or in (the four least-significant bits of) another register, *rY*. The result produce by the `lsl` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*. The last bit shifted-left out of *rX* determines the value of the *c* flag.

The `lsr` instruction performs a logical-shift-right operation (>>). This means that the contents of register *rX* are shifted to the right by the amount specified in *Op2*, and each bit *shifted-in* has the value 0. The effect of this instruction is to perform an *unsigned division* by powers of two. The shift amount is specified in *Op2* in the same way as described previously for the `lsl` instruction. The `lsr` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

The `asr` instruction performs an arithmetic-shift-right operation (>>>). This means that the contents of register *rX* are shifted to the right by the amount specified in *Op2*, and each bit *shifted-in* replicates the *sign-bit* of *rX*. The effect of this instruction is to perform a *signed division* by powers of two. The shift amount is specified in *Op2* in the same way as described previously. The `asr` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

The `ror` instruction performs a rotate-right operation (<<>>). It shifts the contents of register *rX* to the right in a circular fashion, so that each bit shifted out of the least-significant-bit of *rX* is shifted into the most-significant bit. The shift amount is specified in *Op2* in the same way as described previously. The `ror` instruction affects all of the processor's condition-code flags *z*, *n*, and *c*, but the effect on the *c* flag is *undefined*.

Instruction Encodings

Recall from Labs 9 and 10 that instructions are encoded using a 16-bit format. For instructions that have two operands, when *Op2* is a register the encoding is `III0XXX000000YYY`, and when *Op2* is an immediate *#D* constant the format is `III1XXXDDDDDDDDD`. The `ld` and `st` instructions are encoded as `1000XXX000000YYY` and `1010XXX000000YYY`, respectively. You should encode the `pop` instruction similarly to `ld`, with the encoding `1001XXX000000101`. Also, encode `push` similarly to `st`, using the encoding `1011XXX000000101`. Notice that for both `push` and `pop` the `YYY` field is hard-coded to correspond to the stack pointer register, $r5$.

Recall from Lab 10 that the `b{cond}` instruction uses the `XXX` field to encode a *condition*, where `XXX` = 000 (*none*), 001 (*eq*), 010 (*ne*), and so on. Implement the `bl` instruction by using the previously-unassigned code `XXX` = 111.

You should implement the `cmp` instruction similarly to the `add`, `sub`, and `and` instructions. Use the previously-unassigned code `III = 111`; if *Op2* is a register, then `cmp` is encoded as `1110XXX000000YYY`, and if *Op2* is *#D*, then `cmp` is encoded as `1111XXXDDDDDDDDD`. For the shift/rotate instructions you should also use the code `III = 111`, as follows. When *Op2* is a register, encode these instructions as `1110XXX10SS00YYY`, and when *Op2* is *#D* encode them as `1110XXX11SS0DDDD`. In these encodings `SS` specifies the type of shift/rotate, where `SS` = 00 (`lsl`), `01` (`lsr`), `10` (`asr`), or `11` (`ror`). Note that the instruction `cmp rX,rY` and the various shift/rotate instructions share the most-significant digits of their encodings (bits 15-9), which are `1110XXX`. However, for this `cmp` instruction the next six digits (bits 8-3) are `000000`, whereas for the shift/rotate instructions these bits are never all zeros. To differentiate between `cmp rX,rY` and the shift/rotate instructions it is sufficient to examine the digit in bit-position 8.

Barrel Shifter

To implement the required shift and rotate operations for the `lsl`, `lsr`, `asr`, and `ror` instructions, you need to add a 16-bit *barrel shifter* to the processor's ALU. Register *A* should serve as the data input for the barrel shifter and the shift amount should be provided by *Op2*. The FSM has to control the ALU such that its output comes from the barrel shifter when needed, and the FSM has to control the barrel shifter so that it produces the required type of shift, or rotate, operation. Example Verilog code for a barrel shifter is provided in Figure 25. You should augment your ALU using (a modified version of) the code given inside the always block in this module.

```
module barrel (shift_type, shift, data_in, data_out);
    input wire [1:0] shift_type;
    input wire [3:0] shift;
    input wire [15:0] data_in;
    output reg [15:0] data_out;

    parameter lsl = 2'b00, lsr = 2'b01, asr = 2'b10, ror = 2'b11;

    always @(*)
        if (shift_type == lsl)
            data_out = data_in << shift;
        else if (shift_type == lsr)
            data_out = data_in >> shift;
        else if (shift_type == asr)
            data_out = {{16{data_in[15]}},data_in} >> shift;    // sign extend
        else // ror
            data_out = (data_in >> shift) | (data_in << (16 - shift));
endmodule
```

Figure 25: Verilog code for a barrel shifter.

Finite State Machine Timing

To implement each of the new instructions, you will need to augment the finite state machine for your processor. Table 6 indicates how the required signals may be asserted in each time step to implement the instructions in Table 5. Following the style used in Labs 9 and 10, in this table *Select pc* means "put the program counter onto the *Buswires*," *Select #D* means "put the sign-extended immediate data that is in the instruction register (*IR*) onto the *Buswires*," *W_D* means "assert the input to the flip-flop that provides the *write* signal for the memory," and *do_shift* means "set the control signal on the ALU such that its output will be provided by the barrel shifter."

	T_0	T_1	T_2	T_3	T_4	T_5
push	*Select pc*, $ADDR_{in}$, *pc_incr*	*Wait*	IR_{in}	*sp_decr*	*Select rY*, $ADDR_{in}$	*Select rX*, $DOUT_{in}$, *W_D*, *Done*
pop	*Select pc*, $ADDR_{in}$, *pc_incr*	*Wait*	IR_{in}	*Select rY*, $ADDR_{in}$, *sp_incr*	*Wait*	*Select DIN*, rX_{in}, *Done*
bl	*Select pc*, $ADDR_{in}$, *pc_incr*	*Wait*	IR_{in}	*Select pc*, A_{in}, $r6_{in}$	*Select #D*, G_{in}	*Select G*, pc_{in}, *Done*
cmp	*Select pc*, $ADDR_{in}$, *pc_incr*	*Wait*	IR_{in}	*Select rX*, A_{in}	*Select rY* or *#D*, *AddSub*, F_{in}, *Done*	
lsl, lsr asr, ror	*Select pc*, $ADDR_{in}$, *pc_incr*	*Wait*	IR_{in}	*Select rX*, A_{in}	*Select rY* or *#D*, *do_shift*, G_{in}, F_{in}	*Select G*, rX_{in}, *Done*

Table 6: Control signals asserted in each instruction/time step.

Part VIII

You should connect your processor to a memory and I/O devices in the same way as for Lab 10, including the instruction memory, LED, SW, and seg7 (HEX5-0) I/O devices. The design files for this exercise include a suitable top-level file for your use, called *part8.v*, and a new *inst_mem.v* file for the instruction memory. In this design the instruction memory has been increased from the previous size of 256 words to 4K words. Thus, the processor is connected to the memory using 12 address lines, rather than eight. Other than this change, *part8.v* is the same as the top-level file provided in Part V of Lab 10 (*part5.v*).

To assemble code for your processor, you can use the *sbasm.py* assembler. It supports all of the instructions in the processor, including `push`, `pop`, `bl`, `cmp`, `lsl`, `lsr`, `asr`, and `ror`. The Assembler assumes by default that your machine code will not require more than 256 words—to use all of the new 4K memory you have to include the directive

```
DEPTH 4096
```

at the start of your assembly-language program. This directive will cause *sbasm.py* to produce a *memory initialization file* (MIF) that supports up to 4K words of machine code.

Perform the following:

1. First, extend your processor (from Part V of Lab 10) to provide support for subroutines, by implementing the `push`, `pop`, and `bl` instructions. Make sure to change register $r5$ into a counter that has the *up*, *down*, and *load* controls shown in Figure 24. Test your Verilog code by using the Questa or ModelSim Simulator. Sample setup files for the Simulator, including a testbench, are provided along with the design files for this exercise. The sample testbench first resets the processor system and then asserts the *Run* switch, SW_9, to 1. A simple example of assembly language code that can be used to test your subroutine support is given in Figure 26. The first line of code initializes the stack pointer, *sp*, to the value $1000_{16} = 4096_{10}$, which places the stack at the bottom of the 4K memory module. The next line of code in Figure 26 uses a syntax, *=D*, that is supported by the *sbasm.py* assembler for initializing a register with a 16-bit value. The instruction

   ```
   mv    r4, =0x0F0F
   ```

   is implemented by the assembler using the *two* instructions

   ```
   mvt   r4, #0x0F
   add   r4, #0x0F
   ```

   This *=D* syntax can be used as a convenient way of initializing a register to any 16-bit value.

```
START:  mvt    sp, #0x10       // sp = 0x1000 = 4096
        mv     r4, =0x0F0F
        push   r4
        bl     SUBR
        pop    r4
END:    b      END

SUBR:   sub    r4, r4
        mv     pc, lr
```

Figure 26: An assembly-language program to test subroutine support.

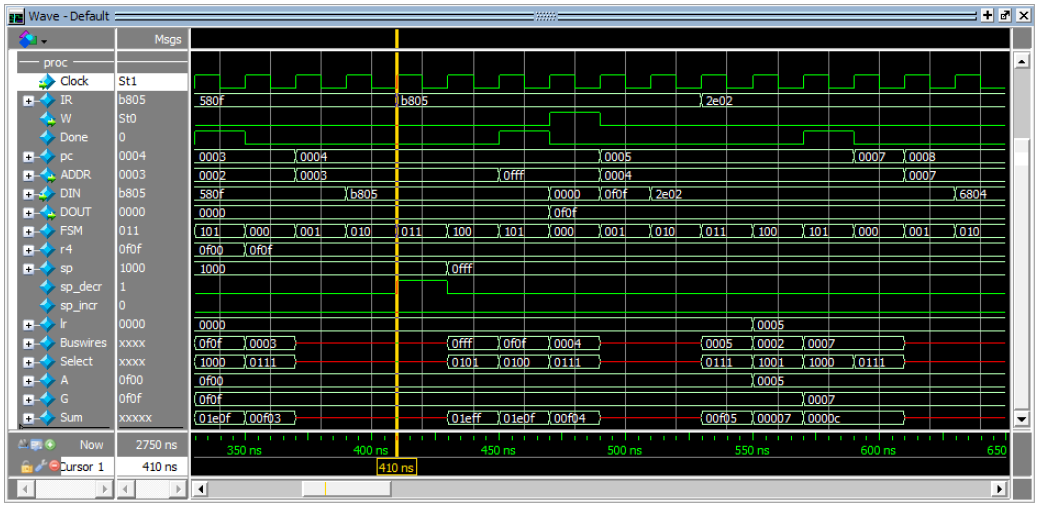

Figure 27: Simulation results for code in Figure 26. (Part a)

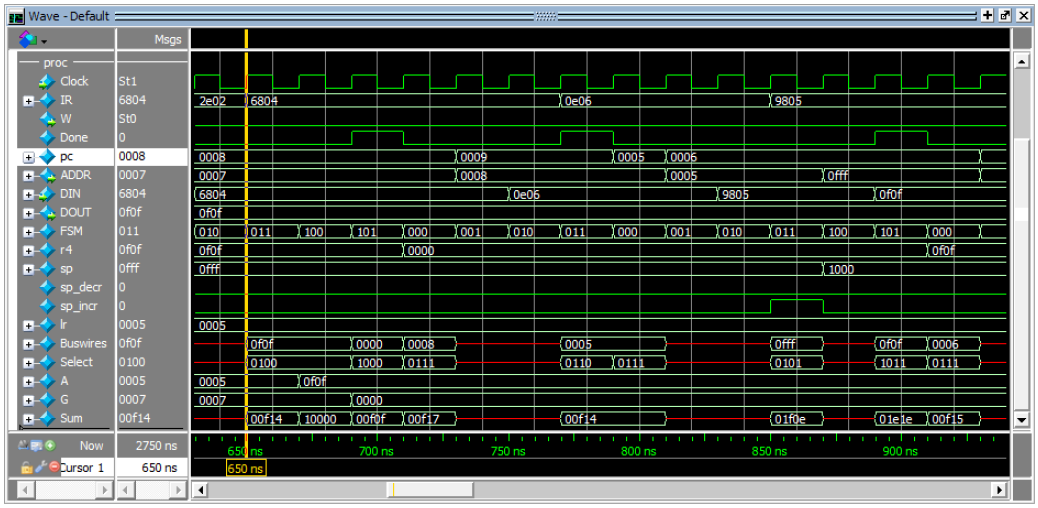

Figure 27: Simulation results for code in Figure 26. (Part b)

An example of simulation results produced by executing the code in Figure 26 is displayed in Figure 27.

In part (a) of the figure the first two lines of code (three instructions) in the program have already been executed, so that the stack pointer *sp* = 0x1000 and register $r4$ = 0x0F0F. At 410 *ns* in simulation time the processor loads the instruction at address 3, which is `push r4` (`0xb805`). As shown in the simulation results the signal *sp_decr* is asserted to decrement *sp* to `0xFFF`, and then the contents of register $r4$ are written to the memory. At 530 ns in simulation time the instruction `bl SUBR` (`0x2E02`) is loaded into *IR*, from address 4. First, this instruction sets the link register *lr* to the value 5 (the subroutine return address), and then sets *pc* = 7, which is the address of `SUBR`.

Figure 27b continues the simulation results from Part (a). The first instruction of the `SUBR` subroutine, `sub r4,r4` (`0x6804`), is loaded into *IR* at 650 *ns*. As shown in the simulation, this instruction results in $r4$ = 0. Then, the subroutine return instruction `mv pc,lr` (`0x0E06`) is executed to return control to the address in *lr*, which is 5. The instruction at address 5 is `pop r4` (`0x9805`). It first reads from the memory at the address in *sp*, which is 0xFFF, and then asserts the *incr_sp* signal, resulting in *sp* = 0x1000. Finally, the data read from the memory is used to restore the value $r4$ = 0x0F0F.

2. Next, you should add the `cmp` instruction, which is similar to `sub`, as well as the shift and rotate instructions. Augment your ALU to include the barrel-shifter capability illustrated in Figure 25. Simulation results for a correctly-designed processor, executing code in Figure 28, are displayed in Figure 29. In Part (a) of the figure the first three instructions in the code have already been executed, so that register $r0$ = 4 and $r4$ = 0x0F0F. At 350 *ns* in simulation time the processor fetches the instruction at address 3, which is `lsl r4, #1`. In time step T_3 of this instruction (which is indicated as `011` in the waveform labeled *FSM*) register $r4$ is placed onto *Buswires* so that it can be copied into register A, in the ALU. Then, in time step T_4 the immediate data, which is in *IR* and specifies the shift amount, is placed onto *Buswires*. The *do_shift* signal is asserted, so that the ALU's *Sum* output is driven by the barrel shifter. It uses bits 3 - 0 from *Buswires* as the shift amount for the `lsl` instruction. The barrel shifter generates the result *Sum* = 0x1E1E, which is loaded into $r4$ at the end of the instruction.

   The next instruction executed in Figure 29a is `lsr r4, #1`. It reverses the previous `lsl` operation, resulting in $r4$ = 0x0F0F. At 590 *ns* in simulation time, the `lsl r4, r0` instruction is executed. Steps $T_0 - T_3$ of this instruction appear in part (a) of Figure 29, and the remaining time steps are shown in Figure 29b. Observe in time step T_4 that register $r0$ is placed onto *Buswires*, because the shift amount (4) is contained in this register. This `lsl` instruction results in $r4$ = 0xF0F0. The final two instructions in the simulation are `asr r4, #1`, which produces $r4$ = 0xF878, and `ror r4, r0`, which results in $r4$ = 0x8F87.

```
START:  mv    r0, #4
        mv    r4, =0x0F0F

        lsl   r4, #1        // lsl with Op2 = #D
        lsr   r4, #1        // lsr with Op2 = #D
        lsl   r4, r0        // lsl with Op2 = rY
        asr   r4, #1        // asr with Op2 = #D
        ror   r4, r0        // ror with Op2 = rY

END:    b     END
```

Figure 28: A program to test shift and rotate instructions.

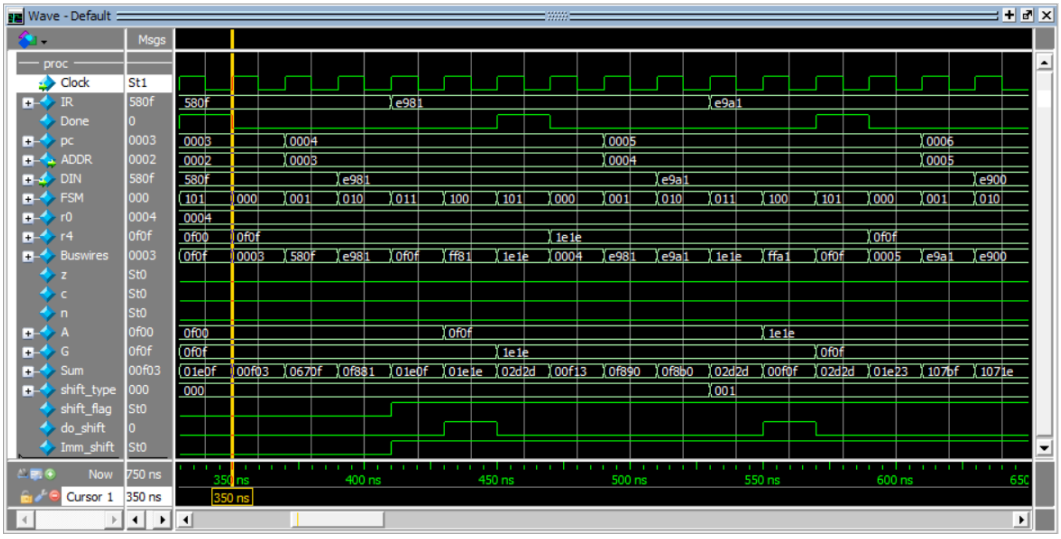

Figure 29: Simulation results for code in Figure 28. (Part a)

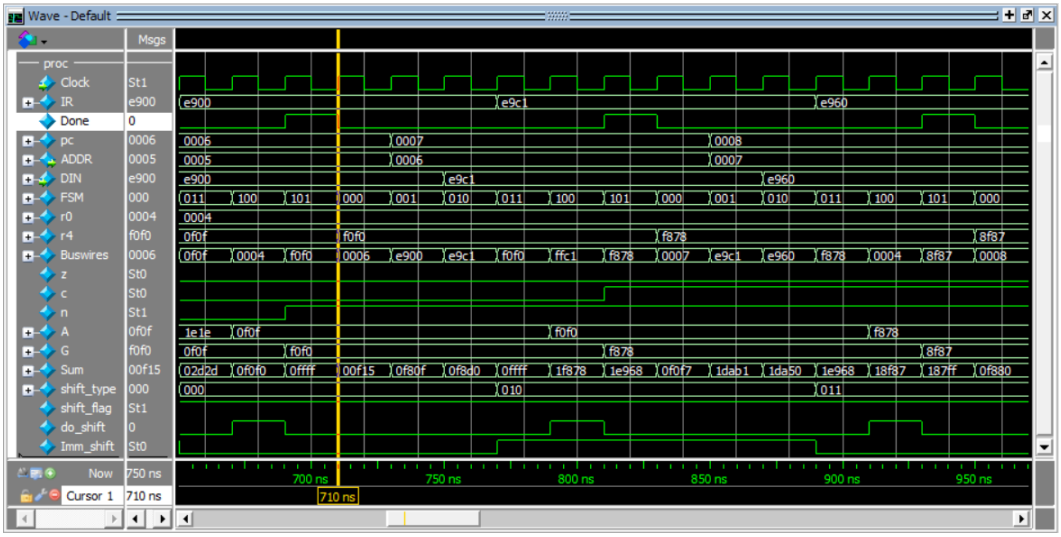

Figure 29: Simulation results for code in Figure 28. (Part b)

3. An example of a subroutine, called `REG`, that you may find useful is given in Figure 30. This subroutine is passed one parameter, in register $r0$. The purpose of the subroutine is to display the contents of this register, in hexadecimal, on HEX3-0. This code utilizes the `push`, `pop`, `cmp`, and `lsr` instructions, and also uses the *lr* to return from the subroutine.

   A main program that calls the `REG` subroutine is provided in Figure 31. This program tests the various shift and rotate operations, as selected by the `SW` switches. The type of shift/rotate is chosen by setting the switches SW_{6-5} = 00 (`lsl`), 01 (`lsr`), 10 (`asr`), or 11 (`ror`). The shift amount is chosen by setting SW_{3-0}. The pattern shifted, $r0$ = 0xF0F0, is loaded at the start of the program. This pattern is reloaded into $r0$ if a shift operation results in $r0$ = 0x0000 or $r0$ = 0xFFFF.

An assembly-language source-code file, called *shift_test.s*, which includes the code in Figures 30 and 31 is provided as part of the design files for this exercise. Assemble this code using *sbasm.py* and ensure that it works with your processor. As mentioned in Labs 9 and 10, you may want to make use of the DESim tool while developing and debugging your processor. A video demonstration of the program in Figure 31 running on a correctly-working processor using the *DESim* tool can be found at the URL:

https://youtu.be/0k5GPGg_Vto

4. Write some assembly-language code of your choosing that demonstrates the operations supported by your processor. You should make use of various I/O devices that are available to your processor, such as the LEDR lights, SW switches, and HEX displays. In general, try to conceive of a program that does something interesting and challenging. You should be able to demonstrate your code working properly on a DE1-SoC, or similar board, but you may want to make use of *DESim* while developing/debugging your code.

```
.define HEX_ADDRESS 0x2000

// subroutine that displays register r0 (in hex) on HEX3-0
REG:    push  r1
        push  r2
        push  r3

        mv    r2, =HEX_ADDRESS  // point to HEX0

        mv    r3, #0            // used to shift digits
DIGIT:  mv    r1, r0            // the register to be displayed
        lsr   r1, r3            // isolate digit
        and   r1, #0xF          // "    "  "  "
        add   r1, #SEG7         // point to the codes
        ld    r1, [r1]          // get the digit code
        st    r1, [r2]
        add   r2, #1            // point to next HEX display
        add   r3, #4            // for shifting to the next digit
        cmp   r3, #16           // done all digits?
        bne   DIGIT

        pop   r3
        pop   r2
        pop   r1
        mv    pc, lr

SEG7:   .word 0b00111111       // '0'
        .word 0b00000110       // '1'
        .word 0b01011011       // '2'
        .word 0b01001111       // '3'
        .word 0b01100110       // '4'
        .word 0b01101101       // '5'
        .word 0b01111101       // '6'
        .word 0b00000111       // '7'
        .word 0b01111111       // '8'
        .word 0b01100111       // '9'
        .word 0b01110111       // 'A' 1110111
        .word 0b01111100       // 'b' 1111100
        .word 0b00111001       // 'C' 0111001
        .word 0b01011110       // 'd' 1011110
        .word 0b01111001       // 'E' 1111001
        .word 0b01110001       // 'F' 1110001
```

Figure 30: A useful subroutine.

```
DEPTH 4096
.define LED_ADDRESS 0x1000
.define SW_ADDRESS  0x3000

START: mv   sp, =0x1000       // initialize sp

MAIN:  mv   r0, =0x9010
       bl   REG               // display r0 on HEX3-0
       bl   DELAY
LOOP:  mv   r1, =SW_ADDRESS
       ld   r1, [r1]
       mv   r2, =LED_ADDRESS
       st   r1, [r2]
       mv   r2, r1
       lsr  r2, #5            // get shift type (SW bits 6:5)

       cmp  r2, #0b00
       bne  LSR
       lsl  r0, r1
       b    CONT
LSR:   cmp  r2, #0b01
       bne  ASR
       lsr  r0, r1
       b    CONT
ASR:   cmp  r2, #0b10
       bne  ROR
       asr  r0, r1
       b    CONT
ROR:   ror  r0, r1

CONT:  bl   REG
       bl   DELAY

       cmp  r0, #0
       beq  MAIN
       cmp  r0, #-1
       beq  MAIN

END:   b    LOOP

// Causes a delay that works well when using DESim. For an actual
// DE1-SoC board, use a longer delay!
DELAY: push r1
       mvt  r1, #0x04       // r2 <- 2^10 = 1024
WAIT:  sub  r1, #1
       bne  WAIT
       pop  r1
       mv   pc, lr
```

Figure 31: A program that test shift/rotate operations.