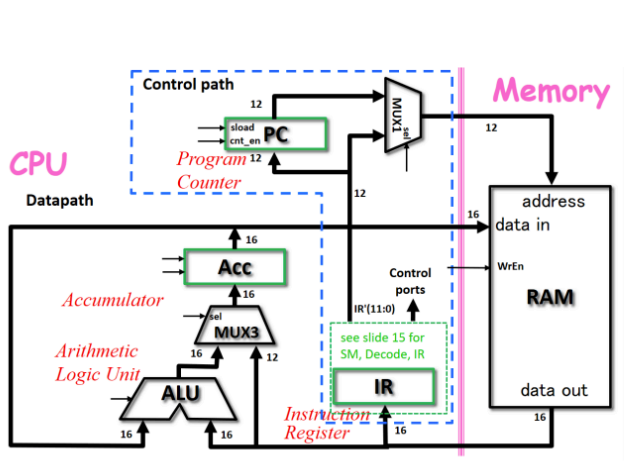
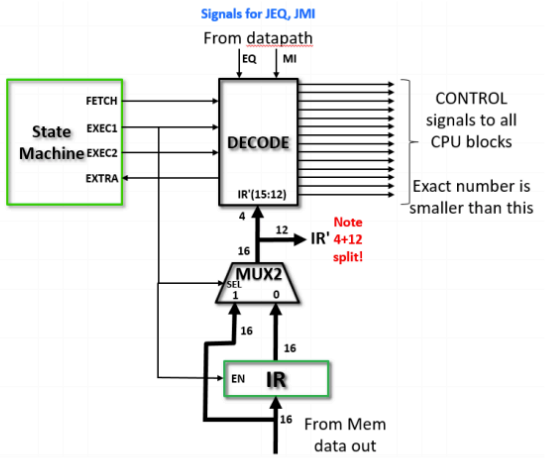
MU0 CPU Design

MU0



slide 5



slide 15

Figure 1: MU0 CPU and memory unit schematics

- Memory/RAM: The place where all initial CPU instructions and data is stored. The RAM is read and written during CPU operation.
- Control Path: The place where the instructions are processed and converted into signals to control all other parts of the CPU.
- Data Path: The place where arithmetical operations take place and data is processed, temporarily stored and modified.

Clock	Symbol	IP	Data ports	Control ports	Function	Name
No	gates/busmux		dataa, datab, result	sel	multiplexer	MUX1, MUX2
Yes	storage/lpm_ff		data, q	sload	DFF register	IR
Yes	arithmetic/ lpm_counter		data, q	cnt_en, sload	loadable counter	PC
Yes		ram 1-port	address, data, q	wren	read/write RAM	RAM

Figure 2: Blocks for Control path: see Figure 3 for control port operation

Device	Input	Operation	1	0
busmux	sel	Select	datab	dataa
lpm_ff	sload	dff load	enabled	disabled
arithmetic/lpm_counter	sload	counter load	enable	disable
	cnt_en	enables +1 count	enable	disable
RAM 1-Port	wren	operation	write	read

Figure 3: Block control signals with operation

MU0 Instructions

LDA, ADD and SUB	LDA, ADD and SUB takes three cycles because the ram_out is the instruction during EXEC1. Ram_out becomes the value in the memory accessed by LDA, ADD or SUB in EXEC2. ALU carries out addition or subtraction and outputs the result to ACC. ACC's value is refreshed after EXEC2.
JMI	PC jumps when Acc < 0.
JEQ	PC jumps when Acc = 0.

Instruction	Binary	HEX
LDA	0b0000	0
STA	0b0001	1
ADD	0b0010	2
SUB	0b0011	3
JMP	0b0100	4
JMI	0b0101	5
JEQ	0b0110	6
STP	0b0111	7
LDI	0b1000	8
LSL	0b1001	9
LSR	0b1010	A

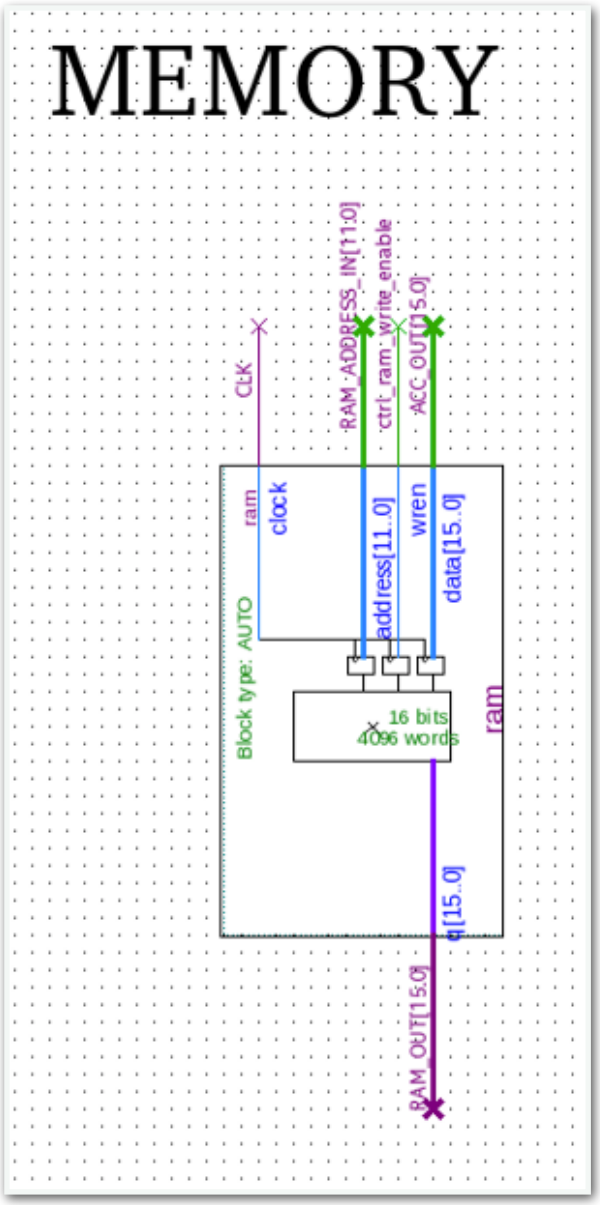
RAM

The Memory block has 4096 memory words, each word being 16 bits wide.

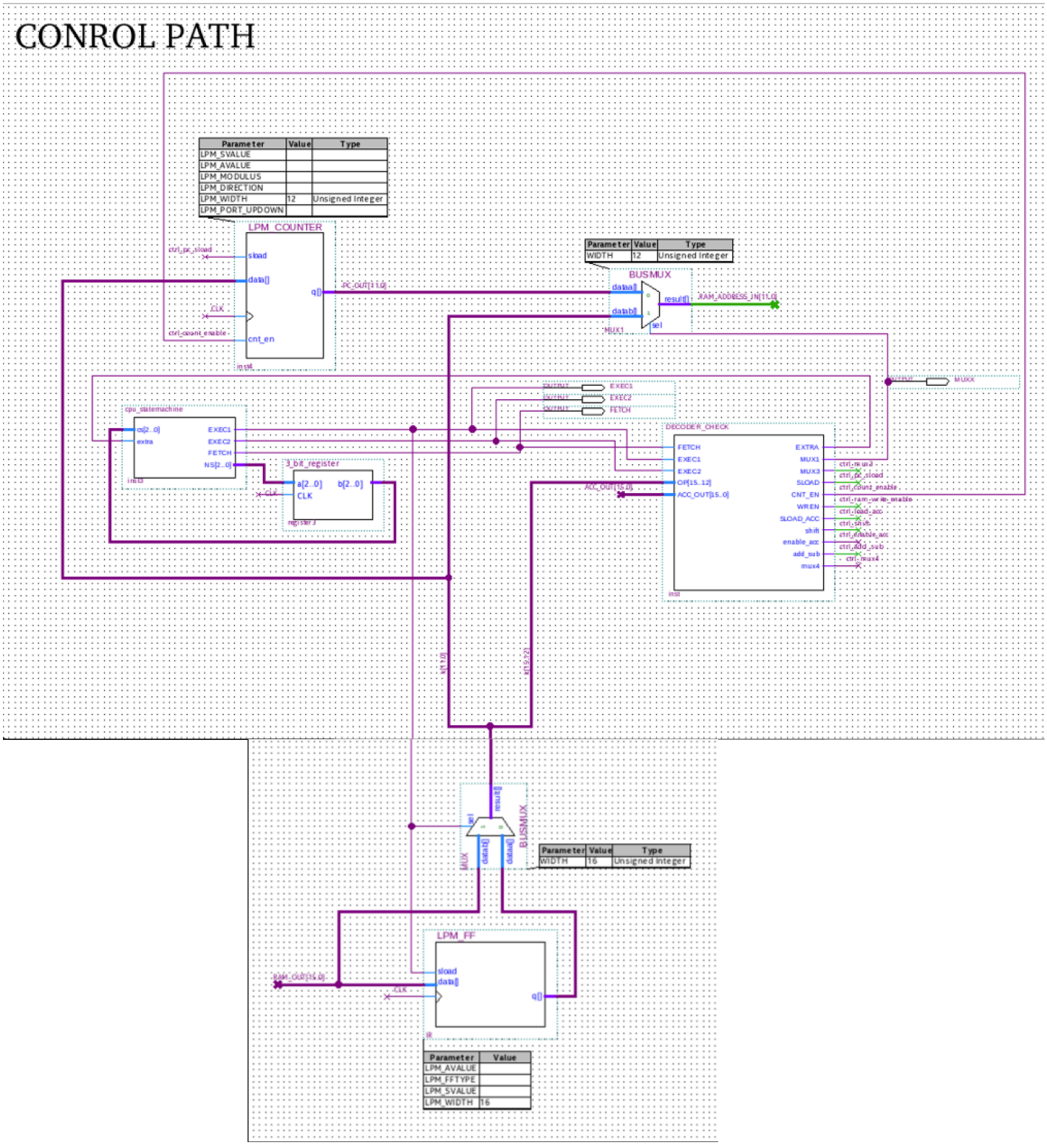
To address the 4096 locations, the RAM requires a 12-bit input address
($log_2(4096) = 12$)

WREN is write enable, which is used during STA to write things into the memory.

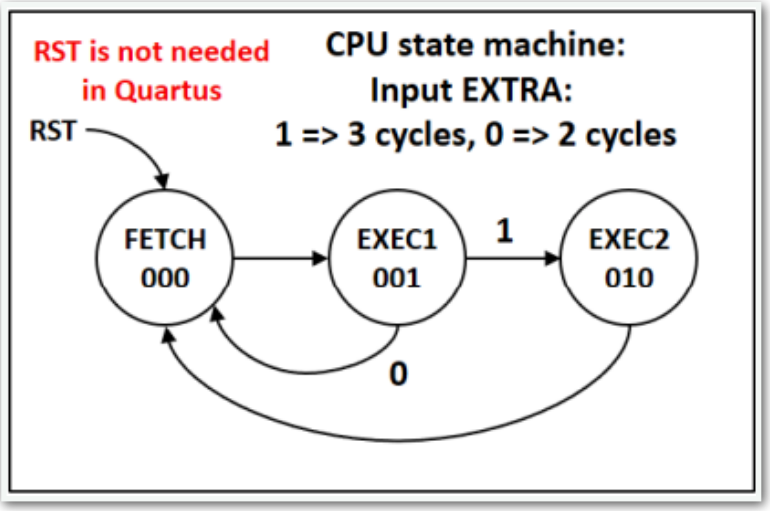
RAM is synchronised by the clock.

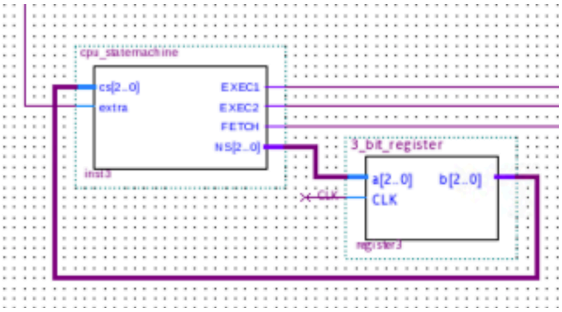


Control Path



State Machine





The state machine synchronously outputs the state, at which the CPU is during a clock cycle.
The state machine takes an EXTRA signal as an input coming from the decoder, which indicates whether the instruction needs 2 or 3 cycles
The state machine outputs are: FETCH, EXEC1, EXEC2

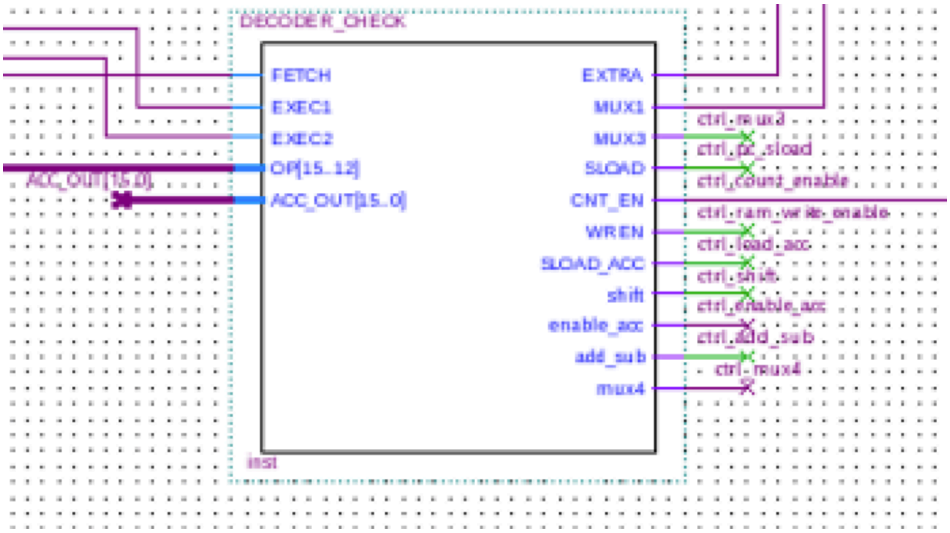
NS (Next State) is clocked by a 3-bit register to become the CS (Current State) during the next clock cycle.

```
module cpu_statemachine
(
     input[2:0] cs,
     input extra,
     output EXEC1, EXEC2 , FETCH,
     output [2:0] NS
);


     assign NS[2] = 0;
     assign NS[1] = (~cs[2]&~cs[1]&cs[0]&extra);
     assign NS[0] = (~cs[0]&~cs[1]&~cs[2]);

     assign FETCH = ~cs[2]&~cs[1]&~cs[0];
     assign EXEC1 = ~cs[2]&~cs[1]&cs[0];
     assign EXEC2 = ~cs[2]&cs[1]&~cs[0];

endmodule
```

Decoder

The decoder is a very important block of the MU0 CPU. It is an asynchronous block containing combinational logic, which outputs various control signals to the rest of the CPU.

The decoder takes the opcode from the instruction word, as well as the execution cycle from the state machine and outputs the following signals:



Decoder I/ O:

```
input FETCH,
input EXEC1,
input EXEC2,
input [15:12] OP,
input [15:0] ACC_OUT,
output EXTRA,
output MUX1,
output MUX3,
output SLOAD,
output CNT_EN,
output WREN,
output SLOAD_ACC,
output shift,
output enable_acc,
output add_sub,
output mux4,
output forcingzero,
output pipeline_enable,
output stop_clock
```

Note: some outputs are only used for the pipelined version of MU0.

The decoder takes the state from the state machine, the opcode from the instruction register and the ACC_OUT from the accumulator in order to output different control signals for an instruction at different states.

Challenge: Add one more instruction to our MU0 design.

We added ASR which is arithmetic shift right. It produces a 1 or 0 at the MSB of the ACC according to the old MSB after shifting right. (0 -> 0, 1->1)

Scaling negatively when right shifting a negative two's complement number.

Un-pipelined Decoder Verilog Combinational Logic

```
wire LDA, STA, ADD, SUB, JMP, JMI, JEQ, STP, LDI, LSL, LSR, EQ, MI, ASR;
        assign LDA = ~OP[15]&~OP[14]&~OP[13]&~OP[12];           // LDA 0000 0
        assign STA = ~OP[15]&~OP[14]&~OP[13]&OP[12];            // STA 0001 1
        assign ADD = ~OP[15]&~OP[14]&OP[13]&~OP[12];            // ADD 0010 2
        assign SUB = ~OP[15]&~OP[14]&OP[13]&OP[12];             // SUB 0011 3

        assign JMP = ~OP[15]&OP[14]&~OP[13]&~OP[12];            // JMP 0100 4
        assign JMI = ~OP[15]&OP[14]&~OP[13]&OP[12];             // JMI 0101 5
        assign JEQ = ~OP[15]&OP[14]&OP[13]&~OP[12];             // JEQ 0110 6

        assign STP = ~OP[15]&OP[14]&OP[13]&OP[12];              // STP 0111 7
        assign LDI = OP[15]&~OP[14]&~OP[13]&~OP[12];            // LDI 1000 8
        assign LSL = OP[15]&~OP[14]&~OP[13]&OP[12];             // LSL 1001 9 (Implement this)
        assign LSR = OP[15]&~OP[14]&OP[13]&~OP[12];             // LSR 1010 A
        assign ASR = OP[15]&~OP[14]&OP[13]&OP[12];              // ASR 1011 B
        assign EQ = ~ACC_OUT[15]&~ACC_OUT[14]&~ACC_OUT[13]&~ACC_OUT[12]&~ACC_OUT[11]&~ACC_OUT[10]&~ACC_OUT[9]
        &~ACC_OUT[8]&~ACC_OUT[7]&~ACC_OUT[6]&~ACC_OUT[5]&~ACC_OUT[4]&~ACC_OUT[3]&~ACC_OUT[2]&~ACC_OUT[1]&~ACC_OUT[0];
        assign MI = ACC_OUT[15];
```

```
assign EXTRA = (LDA | ADD | SUB)&EXEC1;
assign MUX1 = EXEC1&(STA|ADD|SUB|LDA|(JMI&MI)|(JEQ&EQ)|JMP) | EXEC2&(ADD|SUB|LDA);
assign MUX3 = (LDA|LDI) | (EXEC1&(ADD|SUB)) ;
assign SLOAD = (JMP | (JEQ&EQ) | (JMI&MI))&EXEC1;
assign CNT_EN = EXEC2&(LDA|ADD|SUB) | EXEC1&(LDI|STA|LSL|LSR|(JMI&~MI)|(JEQ&~EQ));
assign WREN = STA&EXEC1;
assign SLOAD_ACC = (LDI)&EXEC1 | (SUB|ADD|LDA)&EXEC2;
assign add_sub = ADD;
assign shift = 0;
assign enable_acc = (LDI|LSL|LSR)&EXEC1 | (SUB|ADD|LDA)&EXEC2;
assign mux4 = LDI;
```

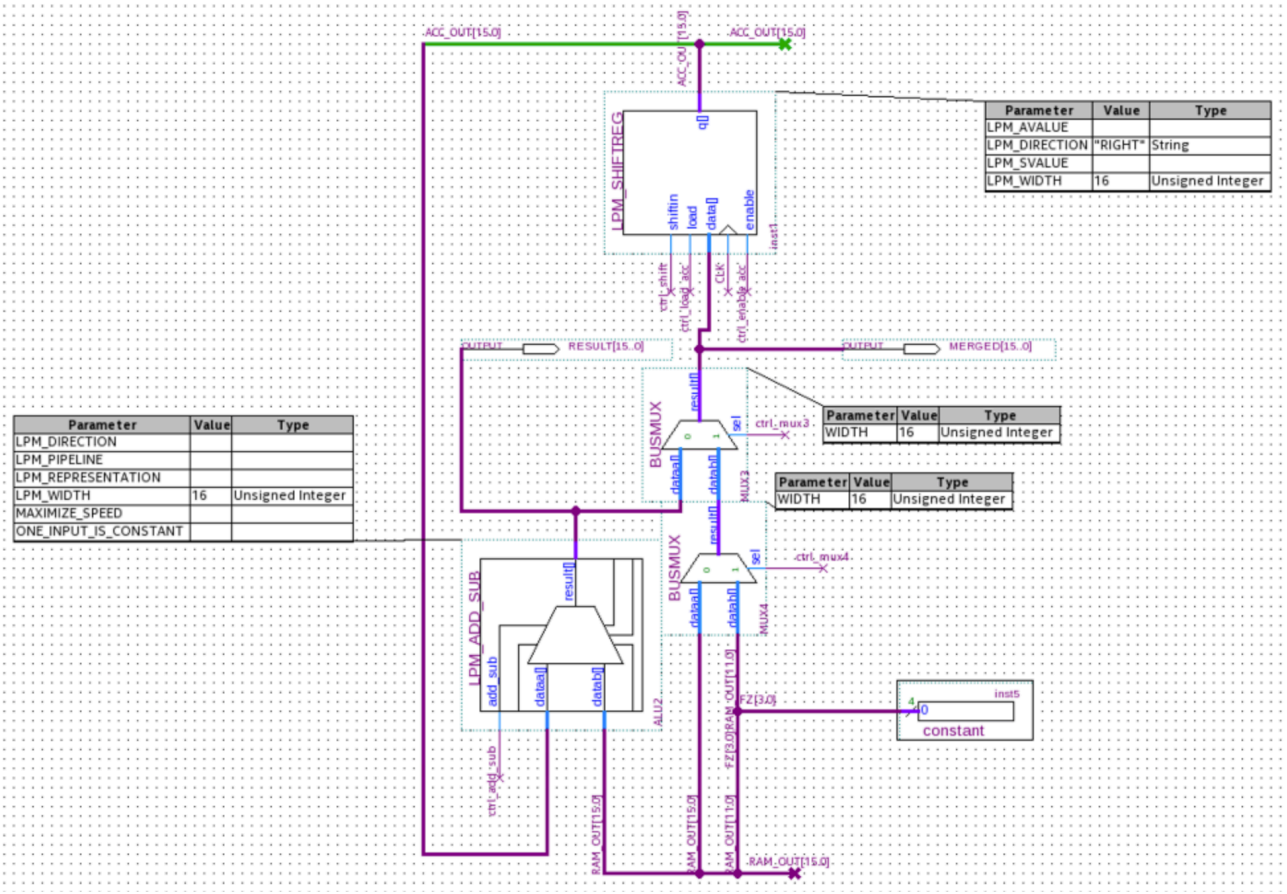
```
endmodule
```

EXTRA	Extra is 1 during EXEC1 of LDA, ADD and SUB, as they are the only three instructions that require three cycles.
MUX1	MUX1 chooses the output of PC when 0, and IR' when 1. *When MUX1 chooses IR', it's mostly at the Exec1 of STA or Exec2 of ADD, SUB or LDA, in order to access a value at a specific memory location.*
MUX3	It chooses IR(11:0) when 1 and the output of the ALU when 0.
CNT_EN	Count_enable enables the PC to count up by one, telling RAM to read the information in the next cell. STP is implemented by not incrementing the PC when having STP instruction.
SLOAD	It loads the PC to a specific number if certain conditions of JMP (no condition), JEQ or JMI are met.
WREN	It allows the RAM to write memory during EXEC1 of STA.
SLOAD_ACC	It allows values to be loaded into Acc.
Enable_acc	It enables shift or load in ACC.
Add_sub	It controls the ALU to implement addition or subtraction.
Shift	It is needed for ASR to write the MSB of the ACC during a right shift.
MUX4	MUX4 is needed for LDI to force the first 4 bits to 0.

Datapath

The data path consists of the ALU, the Accumulator and a multiplexer that selects the input to the accumulator.

All components are synchronous combinational logic, with processes the inputs coming from the control path.



The **accumulator** is a clocked 16-bit wide register block, which can be also be used as a shift register.

The **ALU** is a LPM_ADD_SUB that implements addition or subtraction according to the ctrl_add_sub signal.

MUX4 is used to implement the function of LDI. Since the opcode of LDI is 1000 (8xxx), only the last three-bit hex number should be loaded into the ACC, four forcingzero bits are needed in the front.

For **LDA**, we need to load the full four-bit hex number into the ACC, we do not need the forcing zero bits.

Testing the complete CPU

Professor Clarke's test code provided the following MU0 test-code.

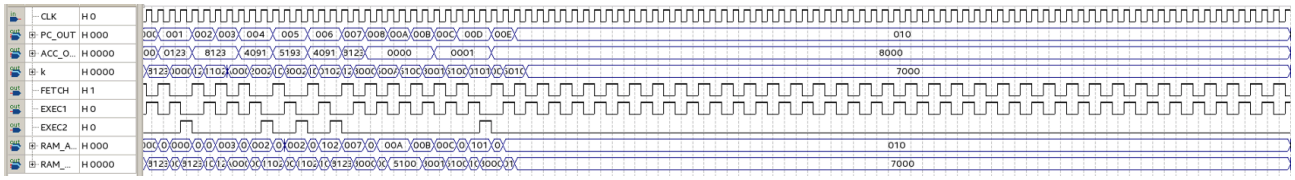
In readable form:

```
0       LDI 0x123 // check Acc = 0x0123 after LDI
1       LDA 0x0   // check Acc =  0x8123 after LDA
2       STA 0x102 // Use STA, can't check yet
3       LSR       // check Acc = 0x4091 after LSR
4       ADD 0x2   // check Acc = 0x4091+0x1102 = 0x5193 after ADD
5       SUB 0x2   // check Acc = 0x4091 after SUB
6       LDA 0x102 // check Acc = 0x8123 after LDA -this checks previous STA
7       LDI 0x0
8       JEQ 10
9       JMP 0x100 // executed if JEQ does not work
10      JMI 0x100 // executed if JMI does not work
11      LDI 1
12      JEQ 0x100 // executed if JEQ does not work
13      LDA 0x101 // load 0x8000
14      JMI 16
15      JMP 0x100 // executed if JMI does not work
16      STP       // 0x10: Jump tests passed
——— initial memory contents ———
        ORG 0x100 // Put code at 0x100
100     STP       // 0x100: Jump tests FAILED
101     DCW 0x8000 // 0x101: constant to test JMI
```

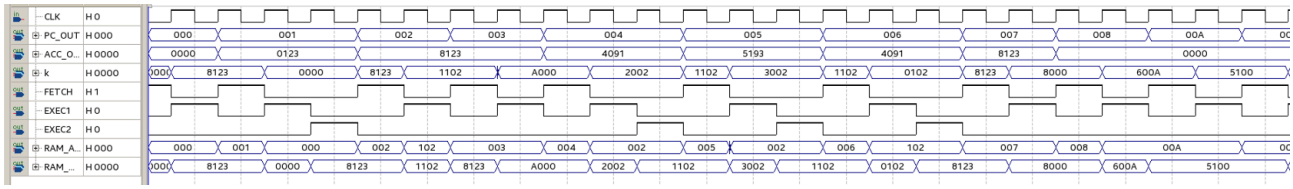
Test-Code entered as HEX into the .mif memory file:

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	8123	0000	1102	A000	2002	3002	0102	8000	.....
8	600A	4100	5100	8001	6100	0101	5010	4100	.....
16	7000	0000	0000	0000	0000	0000	0000	0000	.....
24	0000	0000	0000	0000	0000	0000	0000	0000	.....
32	0000	0000	0000	0000	0000	0000	0000	0000	.....
40	0000	0000	0000	0000	0000	0000	0000	0000	.....
48	0000	0000	0000	0000	0000	0000	0000	0000	.....
56	0000	0000	0000	0000	0000	0000	0000	0000	.....
64	0000	0000	0000	0000	0000	0000	0000	0000	.....
72	0000	0000	0000	0000	0000	0000	0000	0000	.....
80	0000	0000	0000	0000	0000	0000	0000	0000	.....
88	0000	0000	0000	0000	0000	0000	0000	0000	.....
96	0000	0000	0000	0000	7000	8000	0000	0000	.....

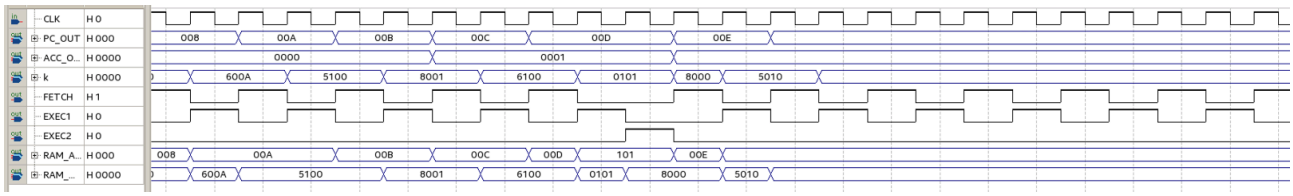
Waveform output:



Detailed:



continued:



Explanation:

In readable form:

```
0       LDI 0x123 // check Acc = 0x0123 after LDI
1       LDA 0x0   // check Acc =  0x8123 after LDA
2       STA 0x102 // Use STA, can't check yet
3       LSR       // check Acc = 0x4091 after LSR
4       ADD 0x2   // check Acc = 0x4091+0x1102 = 0x5193 after ADD
5       SUB 0x2   // check Acc = 0x4091 after SUB
6       LDA 0x102 // check Acc = 0x8123 after LDA -this checks previous STA
7       LDI 0x0
8       JEQ 10
9       JMP 0x100 // executed if JEQ does not work
10      JMI 0x100 // executed if JMI does not work
11      LDI 1
12      JEQ 0x100 // executed if JEQ does not work
13      LDA 0x101 // load 0x8000
14      JMI 16
15      JMP 0x100 // executed if JMI does not work
16      STP       // 0x10: Jump tests passed
——— initial memory contents ———
        ORG 0x100 // Put code at 0x100
100     STP       // 0x100: Jump tests FAILED
101     DCW 0x8000 // 0x101: constant to test JMI
```

After LDI	ACC becomes 0x0123
After LDA	ACC becomes 0x8123
After STA	ACC doesn't change. STA is to be checked later.
After LSR	0x8123 in ACC is shifted right, resulting in 0x4091
After ADD	ACC = 0x4091+0x1102=0x5193
After SUB	ACC = 0x5193 - 0x1102 = 0x4091
After LDA	ACC becomes 0x8123 STA is proven to be working
After LDI	ACC becomes 0x0000. Therefore, the next JEQ loads PC with 10, shown by a gap between 008 and 00A in the PC_OUT.
JMI (00A in RAM) NOT EXECUTED	It is not executed because it only jumps when ACC is a negative number and does not jump when ACC is equal to zero.
After LDI	ACC becomes 0x0001
JEQ NOT EXECUTED	It does not do anything because ACC is not 0x0000 but 0x0001
LDA	It loads 0x8000 into ACC, which is the value stored in the memory 0x101
JMI	It is executed because 0x8000 is a negative number, shown by 010 in the PC_OUT.
STP	It makes PC to rest at 010, indicating a pass of the jump test.

Things to be checked when debugging

Statemachine states:

1. The states should go from Fetch to Exec1 and back to Fetch if there's a two-cycle instruction.
2. The states should go from Fetch to Exec1 to Exec2 and back to Fetch if there's a three-cycle instruction.

PC_OUT:

1. It increments during non-jump or STP instructions.
2. It stops when STP.
3. It loads another number when the conditions of jump instructions are met.

RAM_address_in:

1. RAM_address_in is used to check if the correct ram memory is accessed.
2. It should be PC_OUT during Fetch.
3. It should be the memory required by some other instructions at EXEC1 and EXEC2.

RAM out:

1. After Fetch (during EXEC 1), RAM_OUT should be the instruction.
2. If an instruction requires EXEC2 like LDA, SUB or ADD is carried out, RAM_OUT becomes the value store in the memory which the instruction wants to access.
3. RAM_OUT is don't care during Fetch for all instructions but STA. By checking the value of RAM_OUT during the fetch of another instruction after STA, we can check if STA is functioning correctly.

IR' (k[11:0] in the output wave form):

1. IR' should be the instruction starting from Exec1 of the instruction.
2. IR' is what controls the output of the decoder.

ACC:

1. ACC should be checked at the Fetch of the next instruction to be the right value produced by the last instruction.

MU0 Pipelining

FETCH	*EXEC1 (VF)*		
	FETCH	*EXEC1*	*EXEC2*
FETCH	*EXEC1 (VF)*	*EXEC2*	
	FETCH	*EXEC1*	*EXEC2*

Virtual fetch is added. Decoder and state machine logics are changed. Jumps are not pipelined like STA.

Since during Fetch, the CPU does nothing but read the instruction from the RAM. So we used virtual fetch to replace Fetch where possible, counting up the PC early, causing PC_OUT to read the next instruction early.

PL indicates if an instruction is pipelined.

EXTRA_WIRE indicates whether an instruction requires three cycles.

VF, which is virtual fetch, indicates if the Exec1 or Exec2 in the state machine is overlapped by the fetch of the next instruction, executing the next pipelined instruction one cycle early.

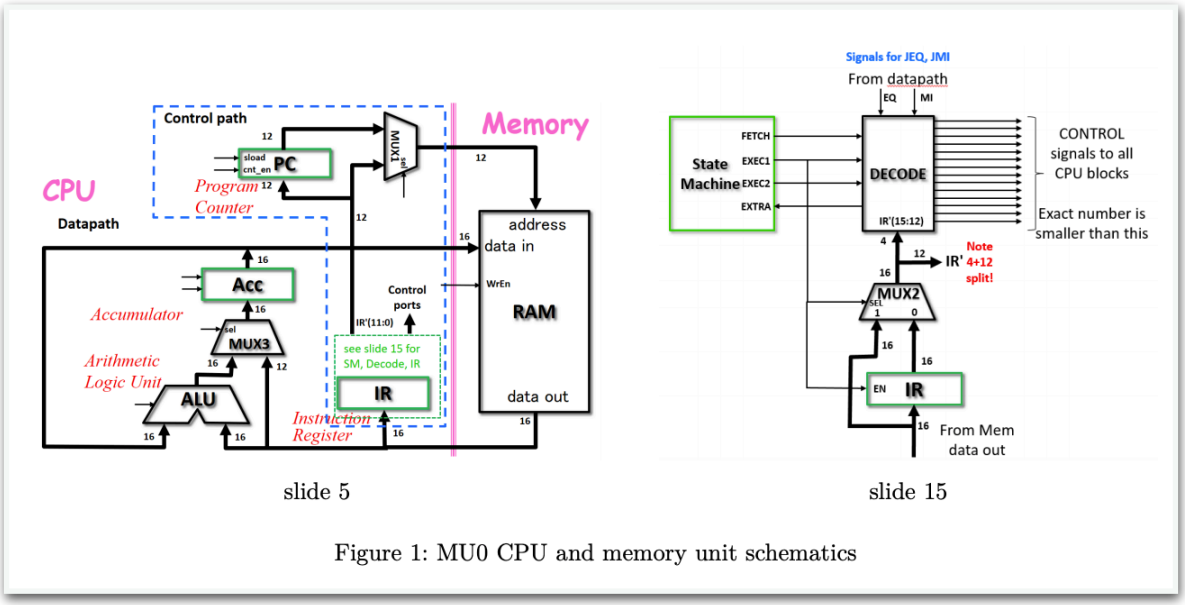


Figure 1: MU0 CPU and memory unit schematics

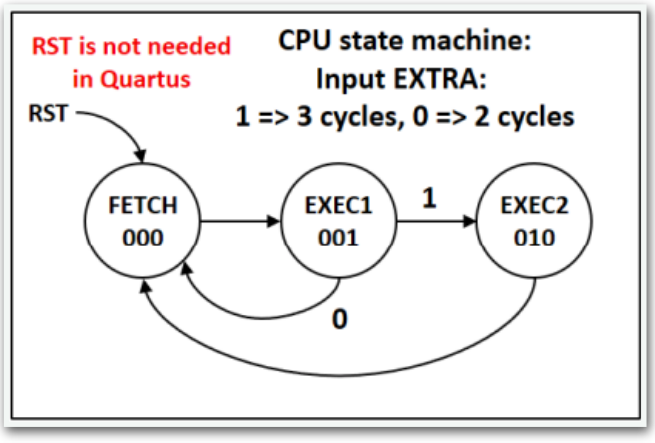
Old State Machine

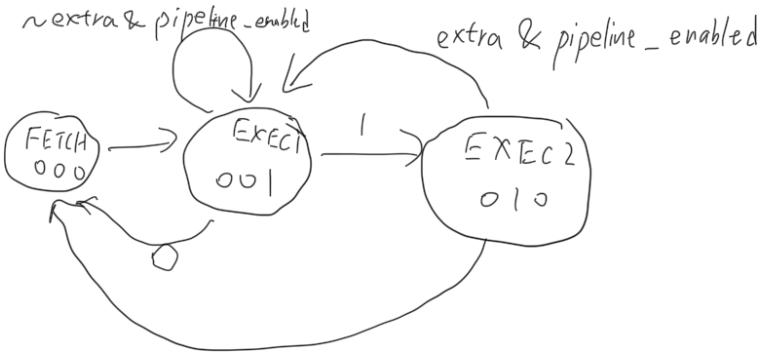
```
module cpu_statemachine
(
      input[2:0] cs,
      input extra,
      output EXEC1, EXEC2 , FETCH,
      output [2:0] NS
);


      assign NS[2] = 0;
      assign NS[1] = (~cs[2]&~cs[1]&cs[0]&extra);
      assign NS[0] = (~cs[0]&~cs[1]&~cs[2]);

      assign FETCH = ~cs[2]&~cs[1]&~cs[0];
      assign EXEC1 = ~cs[2]&~cs[1]&cs[0];
      assign EXEC2 = ~cs[2]&cs[1]&~cs[0];

endmodule
```

New State Machine

```
module cpu_statemachine
(
       input[2:0] cs,
       input extra,
       output EXEC1, EXEC2 , FETCH,
       output [2:0] NS,
       input pipeline_enabled
);


       assign NS[2] = 0;
       assign NS[1] = (~cs[2]&~cs[1]&cs[0]&extra) ;
       assign NS[0] = (~cs[0]&~cs[1]&~cs[2]) | (~cs[2]&~cs[1]&cs[0]&~extra&pipeline_enabled)
                     | (~cs[2]&cs[1]&~cs[0]&extra&pipeline_enabled);

       assign FETCH = ~cs[2]&~cs[1]&~cs[0];
       assign EXEC1 = ~cs[2]&~cs[1]&cs[0];
       assign EXEC2 = ~cs[2]&cs[1]&~cs[0];


endmodule
```

The new state machine takes a new input, pipeline_enabled.





The new decoder logic

```
assign EXTRA_WIRE = (LDA|ADD|SUB);
assign PL = ~STA&~STP&~JMP&~(JEQ&EQ)&~(JMI&MI);
assign VF = PL&(EXEC2&EXTRA_WIRE | EXEC1&~EXTRA_WIRE);


assign EXTRA = (LDA|ADD|SUB)&(EXEC2|EXEC1);
//assign MUX1 = (VF|FETCH)&(STA|ADD|SUB|LDA|(JMI&MI)|(JEQ&EQ)|JMP) | EXEC1&(ADD|SUB|LDA);
assign MUX1 =  ~(FETCH|VF|STP);
assign MUX3 = (LDA|LDI) | (EXEC1&(ADD|SUB)) ;
assign SLOAD = (JMP | (JEQ&EQ) | (JMI&MI))&EXEC1;
assign CNT_EN = (FETCH|VF)&PL | (STA&EXEC1);
//assign CNT_EN = EXEC1&(LDA|ADD|SUB) | (VF|FETCH)&(ASR|LDI|STA|LSL|LSR|(JMI&~MI)|(JEQ&~EQ));
assign WREN = STA&EXEC1;
assign SLOAD_ACC = (LDI)&EXEC1 | (SUB|ADD|LDA)&EXEC2;
assign add_sub = ADD;
assign shift = ASR&EXEC1;
assign enable_acc = (LDI|LSL|LSR|ASR)&EXEC1 | (SUB|ADD|LDA)&EXEC2;
assign mux4 = LDI;
assign stop_clock = STP;

assign pipeline_enable = PL;
```

The changed control outputs:

MUX1	
Function	It chooses the output of PC when 0, and IR' when 1.
Detailed explanation	During fetches and VFs, the MUX1 chooses the output of PC, so that instructions can be pipelined as the correct PC is already there at the beginning of the current instruction. When STP is executed, it stops the PC and produces a 1 for sel_of_MUX1 so that the RAM_ADDRESS_IN is frozen at the instruction which is right before STP.

Pipeline_enable	
Function	It tells the state machine which transition it should carry out.
Detailed explanation	It is used to tell the state machine whether it should go back to Fetch, stay at Exec1 when Exec1, or go to Exec1 when Exec2.

CNT_EN	
Function	It enables the PC to count up by one during a Fetch or a virtual fetch of an instruction that can be pipelined.
Detailed explanation	The first pipelined instruction will increment the PC twice, resulting in pipelining.
Potential Improvements	1. Pipeline jumps. 2. The new CNT_EN logic is not fully correct. ***Correct version:*** ***CNT_EN = (FETCH |*** ***VF)&PL&~STA&~(JMI&~MI)&~(JEQ&~EQ)&~JMP |*** ***EXEC1& Last_instr_is_STAorJumps;*** Last_instr_is_STAorJumps should be the output of a flipflop that stores the value in IR' for one more cycle, and outputs 1 only when the last instruction is STA or Jumps. So that when the last instruction is one that can't be pipelined, the PC counts up at EXEC1 of the current instruction. According to the out CNT_EN logic now, the instruction that is one instructions later than STA (or jumps) will be carried out twice. By changing it to the correct version, it does not count up at the Fetch of the next instruction but the Exec1 of it, producing the correct PC_OUT. The error in the output waveform can be seen as two 004 in RAM_ADDRESS_IN at 50ns and 70ns.

Pipelining Jumps:

Jumps are not pipelined in our design, but it can be implemented by adding a multiplexer after MUX2 in order to replace the value in IR' with RAM_OUT, so that the EXEC1 of the instruction after jump will happen as normal. The selecting line should be controlled by the jumps and be held by a D flip flop to choose RAM_OUT during the exec1 of the instruction after a jump.

If this is not done, IR' will respond the changes in RAM_OUT with 1 cycle delay.

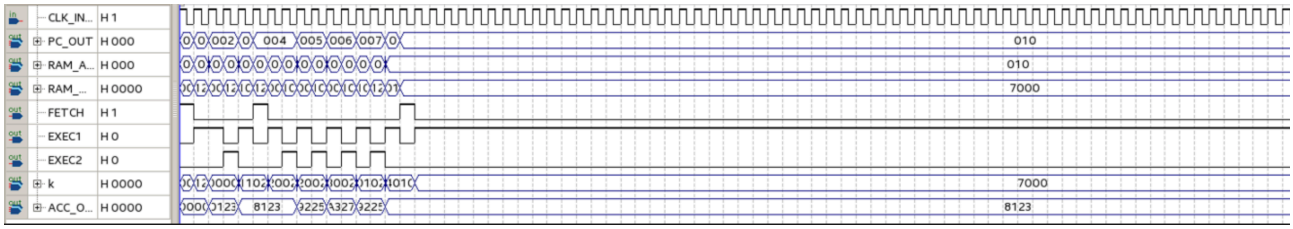
Testing the pipelined CPU

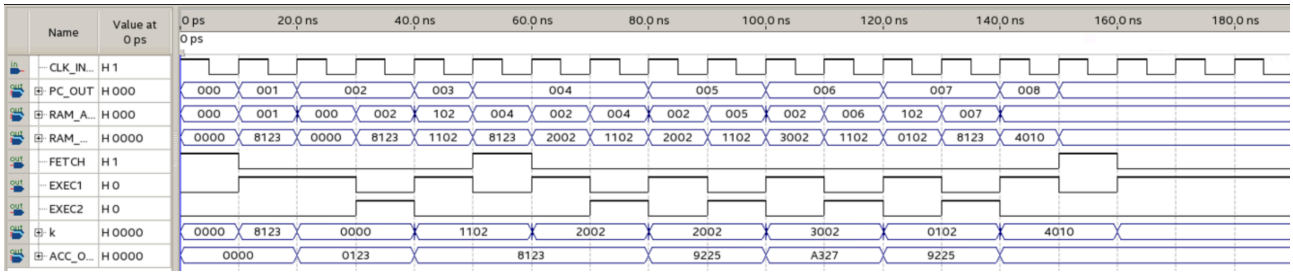
In readable form:

LDI 0x123 // check Acc = 0x0123 after LDI
LDA 0x0 // check Acc = 0x8123 after LDA
STA 0x102 // Use STA, can't check yet
ADD 0x2 // check Acc = 0x8123+0x1102 = 0x9225 after ADD
ADD 0x2 // check Acc = 0x9225+0x1102 = 0xA327 after ADD
SUB 0x2 // check Acc = 0x9225 after SUB
LDA 0x102 // check Acc = 0x8123 after LDA - this checks previous STA
JMP 16
LDI 0x005 // check Acc to be not 5 - this checks previous JMP
——— initial memory contents ———
ORG 0x010// Put code at 0x010
STP

Addr	+0	+1	+2	+3	+4	+5	+6	+7
0	8123	0000	1102	2002	2002	3002	0102	4010
8	8005	0000	0000	0000	0000	0000	0000	0000
16	7000	0000	0000	0000	0000	0000	0000	0000

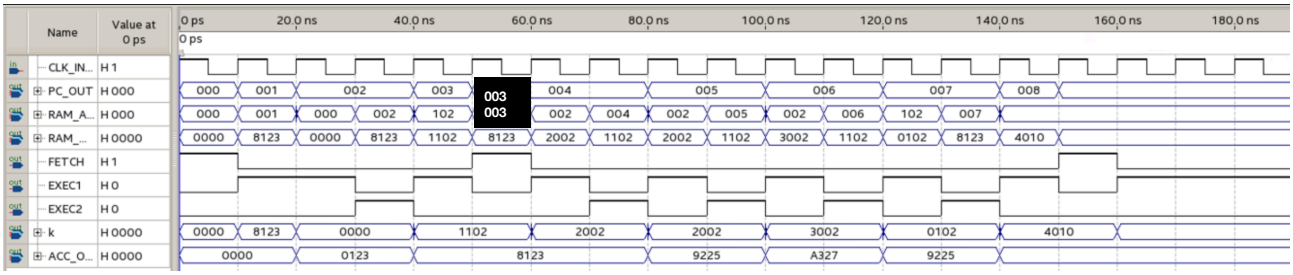
Output waveform





0ns-20ns	LDI 0x123
10ns-40ns	LDA 0x0
30ns-50ns	STA 0x102
50ns-80ns	ADD 0x2
70ns-100ns	ADD 0x2
90ns-120ns	SUB 0x2
130ns-140ns	JMP 16
150ns - 170ns	STP

If the CNT_EN logic was correct, the output waveform would be:



Appendix:
Pipelined statemachine

LDI ADD

The new state machine transits from Fetch to Exec1 first.

Then, it stays at Exec1 for one more cycle because '*extra'* is 0, and pipelining is enabled because LDI is one of the instructions that can be pipelined.

Then, the state machine goes to Exec2 because ADD instruction requires Exec2.

LDA STA LDI

The new state machine transits from Fetch to Exec1 and to Exec2.

Since LDA produces a 1 in 'extra', and pipelining is enabled because LDA is one of the instructions that can be pipelined, the state machine goes from Exec1,

After Exec1, the state machine returns to Fetch because STA produces a 0 in pipeline_enabled.

The state machine then moves from Fetch to Exec1 to carry out LDI.

Note: *in example 2, if STA is replaced by JMP, JMI or JEQ. The same transitions of the state machine will happen.*