### 



**MS2  
Project 1**

Andrew Nady

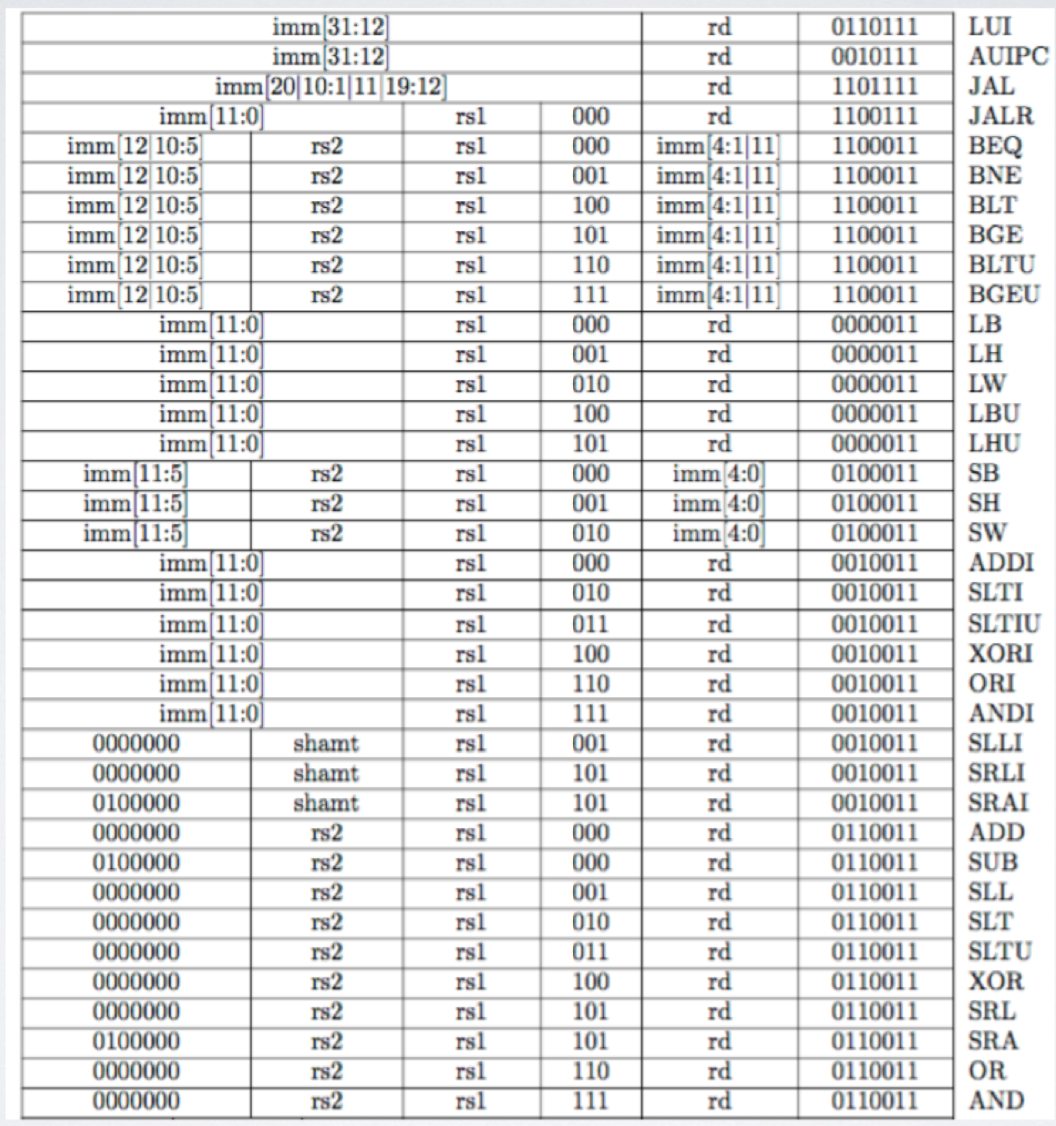
Salma Soliman

### **RISC-V-Single-Cycle-Microprocessor**

Introduction:

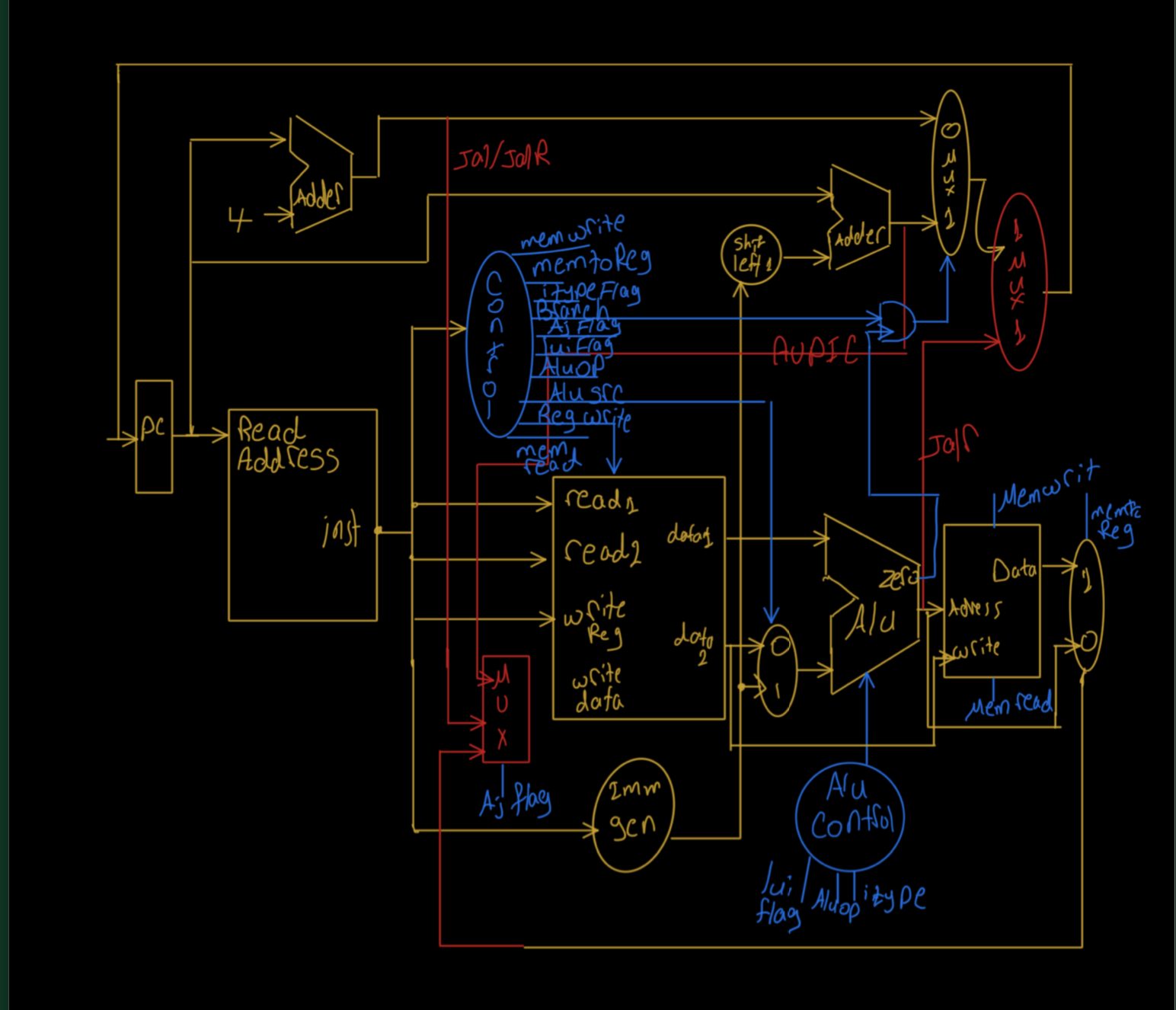
We are designing a RISC-V processor that supports the base integer instruction set according to the specifications found here: <https://riscv.org/technical/specifications/>

We implemented each instruction according to the specification found in page 130 of the RISC-V Instruction Set Manual –Volume I: Unprivileged ISA and explained in Chapter2 of the same manual.



Design:

* To support JAL/JALR/AUIPC, we added a MUX that’s colored in red in the below pic.
* To support JALR, we added an additional MUX that chooses between the output of the previous mux and the JALR signal.
* We added other singles in the control unit in order to differentiate between some new instructions
  + We added the Aj flag which distinguishes the Auipc and the jal instructions from each other where Aj flag selects from the jal/jalr, Auipc or from the alu
  + We added an i\_type flag where some instructions such as addi can make conflict between sub and addi so we make this flag to differentiate between them.
  + We added lui flag which tells the ALU to shift left the immediate 12 times.



Algorithm:

In order to support the whole instructions, we did that by dividing them into groups (R-type, I-type,load instruction group, store instruction group, branches group, jal,jalr and AUIPC).

For supporting R- type instructions:

Based on the alu\_selection line we choose the output of the ALU to be the result of the operation it does.

|  |
| --- |
| **case**(aluSel)  **4'b0000**: result = sum;  **4'b0001** :result= sub ;  **4'b0101** : result= in1 & in2 ;  **4'b0100** : result= in1 | in2 ;  **4'b0111**:result=in1^in2;  **4'b1000**: result = in1>>in2; //SRL  **4'b1010**:result = in1>>>in2; //SRA  **4'b1001**:result = in1<<in2; //SLL  **4'b1101**:result={(in1<in2)?**1**:**0**};//SLT  **4'b1111**:result={({in1<**0**?-**in1:**in1}<{in2<**0**?-**in2:**in2})?**1**:**0**};//SLTU  **4'b0110**:result={in2,**12'b0**}; ///LUI |

For supporting I- type instructions:

Based on a flag we created called (I\_type flag ). As in the I-type instructions , the processor performs the same operations as R-type instructions and the only difference is the second operand where the ALU takes an immediate instead of a register operand. This flag is only activated in the control unit when the opcode equals the I-type opcode and deactivated for the rest of the instructions. Furthermore, in the ALU control unit we distinguished between the instructions using this flag.

|  |
| --- |
| **else** **if** (aluop == **2'b10** &&lui\_flag==**0** &&  i\_type == 0&& instr2 == **0** && intr1 == **3'b000**) // R- type instruction (ADD) //instr2 VI  aluS = **4'b0000**;  **else** **if** (aluop == **2'b10** &&lui\_flag==**0** && i\_type == 1 && intr1 == **3'b000**) // R- type instruction (ADDi) //same aluS as add  aluS = **4'b0000**; |

For supporting B-type instructions:

For supporting the branching, we used the zero flag as an indication that the argument of any branch is true or false. Once it’s true, the zero flag is set to 1, otherwise 0. For the unsigned branches, we settled on the sign first. If the number is negative, we multiply it by -1 and then compare. If it’s positive, we compare directly.

|  |
| --- |
| **4'b0010** : // branching  **begin**  **case**(func3)  **3'b000**: **if** (in1 == in2) zero =**1**; **else** zero = **0**; // beq zero =1 branch =1  **3'b001**: **if** (in1 != in2) zero = **1**; **else** zero = **0**; // bne branch =1 zero =0  **3'b100**: **if** (in1<in2) zero = **1**; **else** zero = **0**; // blt branch=1  **3'b101**: **if** (in1>= in2) zero = **1**; **else** zero = **0**; //bge branch=1  **3'b110**: **if** ({in1<**0**?-**in1:**in1}<{in2<**0**?-**in2:**in2}) zero = **1**; **else** zero = **0**; // bltu branch=1  **3'b111**: **if** ({in1<**0**?-**in1:**in1}>={in2<**0**?-**in2:**in2}) zero = **1**; **else** zero = **0**; //bgeu branch=1 |

For supporting Store-type instructions:

We modified the data memory module by passing function 3 for it so that it could distinguish between instructions. In order to support s-type, we need to activate the memwrite signal coming from the control unit and deactivate memread.

|  |
| --- |
| **if** (MemWrite==**1** && func3 ==**3'b010**) //SW  mem[addr] <= data\_in;  **else** **if** (MemWrite==**1** && func3 ==**3'b001**)//sh  mem[addr][**15**:**0**] <= data\_in;  **else** **if** (MemWrite==**1** && func3 ==**3'b000**)//sb  mem[addr][**7**:**0**] <= data\_in;  **end** |

For supporting load-type instructions:

In order to support U-type, we need to activate the memread signal coming from the control unit and deactivate memwrite.

|  |
| --- |
| **begin**  **if** ( func3 == **3'b000**) //LB  data\_out = MemRead ? mem[addr][**7**:**0**]: **0**;  **else** **if** ( func3 == **3'b001**) //LH  data\_out = MemRead ? mem[addr][**15**:**0**]: **0**;  **else** **if** ( func3 == **3'b010**) //LW  data\_out = MemRead ? mem[addr]:**0**;  **else** **if** ( func3 == **3'b100**) //LBU  data\_out = MemRead ?{ **24'h0**,mem[addr][**7**:**0**]}:**0**;  **else** **if** ( func3 == **3'b101**)// LHU  data\_out = MemRead ? {**16'h0**,mem[addr][**15**:**0**]}: **0**; |

For supporting JAL instruction:

We activated the branch signal in the control unit to allow the pc to branch. Also, we have added a new signal called AJ. it’s a 2 bit signal. The first bit indicates the AUIPC signal and the second bit indicates the J signal related to jal/jalr.

|  |
| --- |
| **5'b11011**: //jal  **begin**  branch=**1**;  memread=**0**;  memtoreg=**0**;  aluop=**11**;  memwrite=**0**;  alusrc=**1**;  regwrite=**0**;  i\_type=**0**;  lui\_flag=**0**;  AJ\_control=**2'b01**; |

In the ALU control unit, we added this code that chooses the selection line.

|  |
| --- |
| **if**(aluop==**2'b11**) //jal  aluS=**4'b0011**; |

We activate the zero flag to branch in the ALU module.

|  |
| --- |
| **4'b0011**:zero=**1**; |

For supporting JALR instruction:

|  |
| --- |
| **5'b11001**: //jalr  **begin**  branch=**0**;  memread=**0**;  memtoreg=**0**;  aluop=**10**;  memwrite=**0**;  alusrc=**1**;  regwrite=**1**;  lui\_flag=**0**;  i\_type=**0**;  AJ\_control=**2'b01**; |

For supporting AUIPC instruction:

|  |
| --- |
| **5'b00101**: //auipc  **begin**  branch=**0**;  memread=**0**;  memtoreg=**0**;  aluop=**00**; //add as load  memwrite=**0**;  alusrc=**1**;  lui\_flag=**0**;  regwrite=**1**;  i\_type=**0**;  AJ\_control=**2'b11**; |

For supporting LUI instruction:

In the ALU module we added this code that loads the register with the upper 20 bits of the immediate and the rest of bits set them to zero.

|  |
| --- |
| **4'b0110**:result={in2,**12'b0**}; ///LUI |

In the control unit we added this code.

|  |
| --- |
| **5'b01101**: //lui  **begin**  branch=**0**;  memread=**0**;  memtoreg=**0**;  aluop=**10**;  memwrite=**0**;  alusrc=**1**;  regwrite=**1**;  i\_type=**0**;  lui\_flag=**1**;  AJ\_control=**2'b00**; |

Testing:

We created 4 assembly files in order to test our single cycle processor

R type

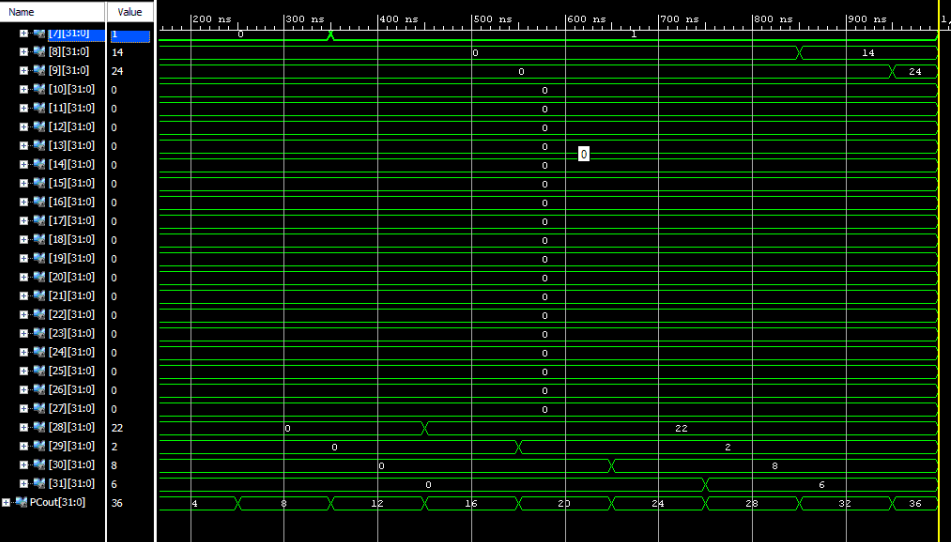
1) we created R\_typeTest.asm in order to test all the R type instructions,

2) then we converted them to binary to write them inside the instruction memory module

|  |  |
| --- | --- |
| Assembly | Binary |
| lw t0, **0**(zero) # **10** ->**1010**  lw t1, **4**(zero) #**12** ->**1100**  lw t2, **8**(zero) #**1** ->**5**  add t3,t1,t0 #**22**  sub t4,t1,t0 #**2**  **and** t5,t1,t0 #**1000** ->**8**  **xor** t6,t1,t0 #**0100** ->**4**  **or** s0,t1,t0 #**1110** ->**14**  sll s1,t1,t2 #**24**  slt s2,t1,t0 #**0**  sltu s3,t1,t0 #**0**  sra s4,t1,t2 #**6**  srl s5,t1,t0 #**6** | 00000000000000000010001010000011  00000000010000000010001100000011  00000000100000000010001110000011  00000000010100110000111000110011  01000000010100110000111010110011  00000000010100110111111100110011  00000000010100110100111110110011  00000000010100110110010000110011  00000000011100110001010010110011  00000000010100110010100100110011  00000000010100110011100110110011  01000000011100110101101000110011  00000000010100110101101010110011 |

Screenshot

register

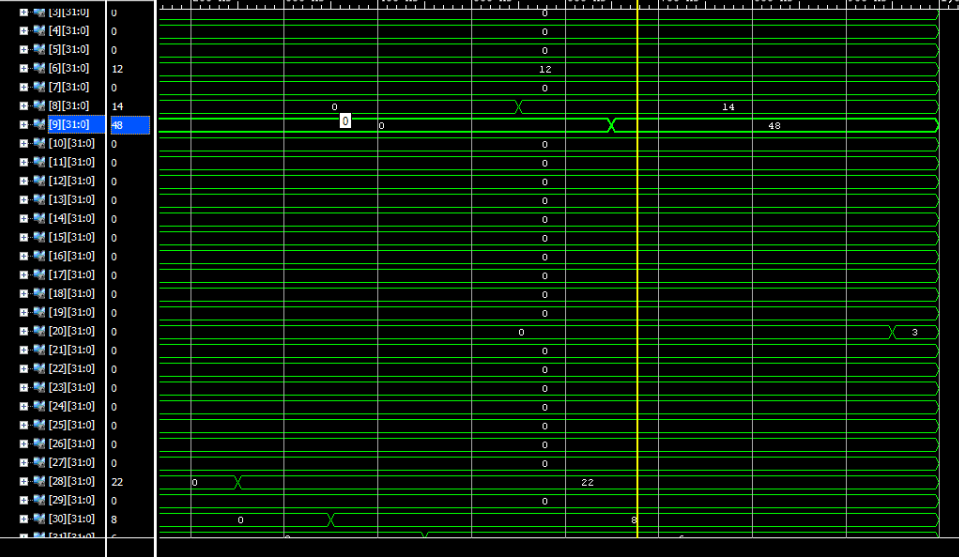


I type

|  |  |
| --- | --- |
| Assembly | Binary |
| lw t1, **4**(zero) #**12** ->**1100**  addi t3,t1,**10** #**22**  andi t5,t1,**10** #**1000** ->**8**  xori t6,t1,**10** #**0100** ->**4**  ori s0,t1,**10** #**1110** ->**14**  slli s1,t1,**2** #**24**  slti s2,t1,**10** #**0**  sltiu s3,t1,**10** #**0**  srai s4,t1,**2** #**6**  srli s5,t1,**10** #**6** | 00000000010000000010001100000011  00000000101000110000111000010011  00000000101000110111111100010011  00000000101000110100111110010011  00000000101000110110010000010011  00000000001000110001010010010011  00000000101000110010100100010011  00000000101000110011100110010011  01000000001000110101101000010011  00000000101000110101101010010011 |

Screenshot

Registers

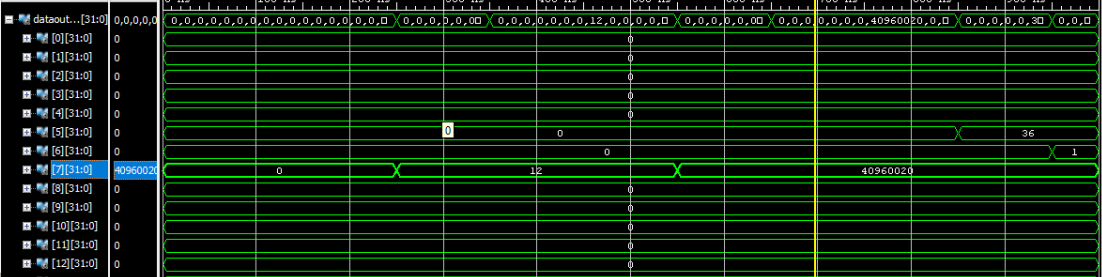


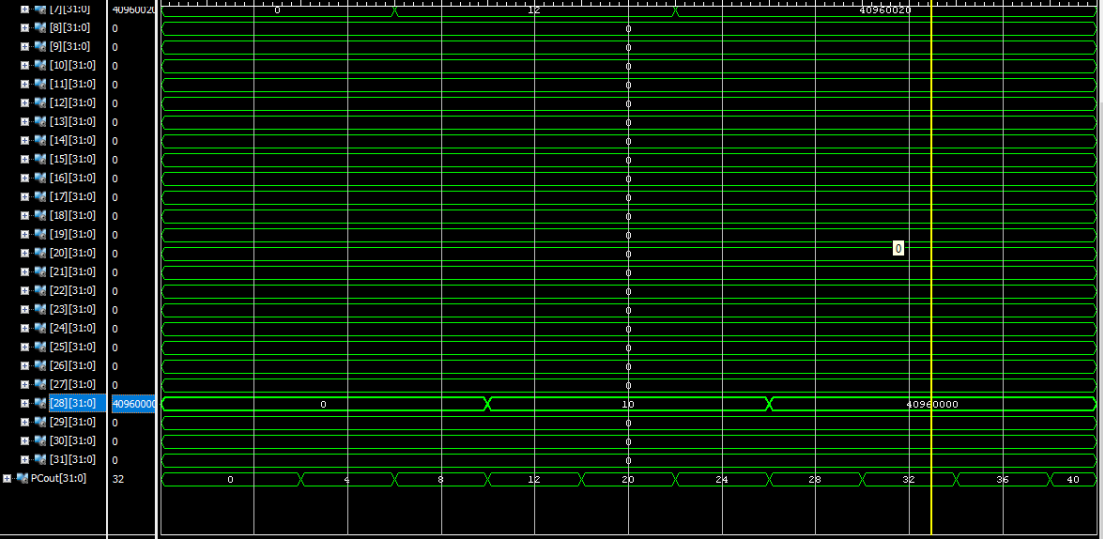
U type&jalr&jalr

|  |  |
| --- | --- |
| Assembly | Binary |
| lw t1,**0**(zero)#**0**  lw t2,**4**,(zero) #**12**  lw t3,**8**(zero) #**10**  jal jumb  addi t1,t1,**1**  **jumb:**  auipc t2,**10000**  lui t3,**10000**  addi t4,zero,**0**  jalr t0,t4,**36**  addi t1,t1,**1**  **jumb2:**  sw t3,**12**(zero) #**10** pc=**36** | 00000000000000000010001100000011  00000000010000000010001110000011  00000000100000000010111000000011  00000000100000000000000011101111  00000000000100110000001100010011  00000010011100010000001110010111  00000010011100010000111000110111  00000000000000000000111010010011  00000010010011101000001011100111  00000000000100110000001100010011  00000001110000000010011000100011 |

Screenshot

Registers and PC





S type

|  |  |
| --- | --- |
| Assembly | Binary |
| lw t1,**0**(zero)#**0**  lw t2,**4**,(zero) #**12**  lw t3,**8**(zero) #**10**  sw t1,**12**(zero)#**0**  sh t2,**16**,(zero) #**12**  sb t3,**18**(zero) #**10** | 00000000000000000010001100000011  00000000010000000010001110000011  00000000100000000010111000000011  00000000011000000010011000100011  00000000011100000001100000100011  00000001110000000000100100100011 |

Screenshot

Memory:

