344 RISCV Datapath Simulator Report

Contents

1) Input File Format

Valid instructions are *add, sub, and, or, ld, sd, beq*. All instructions have 3 operands. Also there are labels.
Invalid inputs are not accepted. There could be whitespaces between operands or empty lines.

Instruction format:		**Label format:**
add || sub || and || or	r1,r2,r3	<label> :
ld || sd	r1, offset(r2)	
beq	r1,r2, label	

2) Components

2.1) Instruction Memory

This class takes an input *file path* and a *PC pointer* as parameters to its constructor. It reads instructions one by one from the given file and stores them after tokenizing the instructions. An instruction is stored in vector<string>, all instructions are stored in a vector<vector<string>>.
Also there is a label map keeping labels with their addresses.

void readInstruction(vector<string>& inst):

Reads the instruction from instructions vector from the index PC points. If the instruction is a label, then it will read the next instruction. If the instruction is "beq", it will calculate and read the offset instead of label.

2.2) Register File

This class represents all units that consist of registers: Register file, intermediate registers (IF/ID,ID/EX,EX/MEM,MEM/WB) and PC.
Its constructor takes a *size* and *regWrite* control signal. Size is the number of registers this object stores. For instance, the PC has 1 register and the register file has 32 registers.
A long vector (64 bit) of specified size is initialized to 0.
regWrite signal controls if a value can be written to this object. This is always true unless the object is not the register file. In the register file case, the value of the regWrite depends on the instruction. regWrite should be set to *false* in the constructor.
Since registers can be updated only at the rising edge of the clock cycle, there is a ***temporary*** *register vector* that stores the calculations during the clock cycle and there is a *register vector* that stores actual values of the registers. Actual values are the values written at the last rising edge.

void update():

This method writes all temporary values, calculated during clock cycle, to the registers. This method is called at the rising edge of the clock.

Other methods are get and set methods. Any register value can be read and set via these methods.

void getReg(long& reg1, int indx1),

long getReg(int indx1),

void getReg(long& reg1, long& reg2, int indx1, int indx2):

These methods are used to read register values at the given indexes. They read the actual values of the registers.

void setRegWrite(bool regWrite):

The value of the regWrite can be set.

void setReg(int regIndx, long data):

Sets the value of the register at given index to the given data. It writes to the temporary vector, since register values are calculated and set during clock cycles.

2.3) Control Unit

Sets all control signals (regWrite, aluSrc, memRead, memWrite, memToReg, pcSrc, aluOp) according to the given instruction.
The control signals can be read via getters.
All instructions have their corresponding integer codes:

add -> 0	sub -> 1	and-> 2	or-> 3
ld -> 4	sd -> 5	beq ->6	

Even though not explicitly stated, when the instruction number is set to a number other than these 7, the control unit makes all control signals 0, so instruction acts as ***nop*** instruction.
Control signals set for instructions:

	regWrite	aluSrc	memRead	memWrite	memToReg	pcSrc	aluOp
add	1						0
sub	1						1
and	1						2
or	1						3
ld	1	1	1		1		0

sd		1		1			0
beq							1

void setOperation(int operation):
Sets the operation whose control values will be calculated.

bool getRegWrite(), **bool** getAluSrc(), **bool** getMemRead(), **bool** getMemWrite(), **bool** getMemToReg(), **bool** getPcSrc(), **int** getAluOp():
Returns the corresponding control signal based on the operation.

void fillReg(RegisterFile* regs, int idx):
Given a register file and index, fills the control signals to the register file starting from *idx*. Order of the control signals can be found. Used to fill the id_ex registers based on the instruction.

2.4) Alu

Represents the arithmetic logic unit. Takes two inputs (*input_1* and *input_2*) and an operation. Possible operations with their corresponding integer codes are:
add -> 0 sub -> 1 and -> 2 or -> 3
The corresponding codes for the instructions are determined by the control unit and sent via AluOp.

void setOperation(int operation):
Sets the operation of the alu.

void setInput_1(int input_1), **void** setInput_1(int input_2), **void** setInput(int input_1, int input_2):
Sets the inputs of the alu.

int getOutput():
Returns the arithmetic result based on its operation and inputs.

int getOutput(int operation):
Sets the operation to the given parameter, returns the output.

int getOutput(int input_1, int input_2, int operation):
Sets the operation, input_1 and input_2 to the given parameters, returns the output.

2.5) Mux

Mux represents the multiplexer. It has two inputs named *input_1* and *input_2*, and a select.

void setSelect(bool select):

Sets the select field of the mux.

void setInput_1(int input_1), **void** setInput_1(int input_2), **void** setInput(int input_1, int input_2):

Sets the inputs of the mux.

int getOutput():

Returns one of the inputs based on select. If select is set to 0, *input_1* is returned; if select is set to 1, *input_2* is returned

int getOutput(bool select):

Sets the select to the given parameter, returns the output.

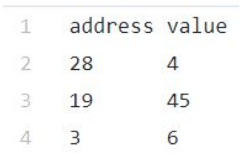
int getOutput(int input_1, int input_2, bool select):

Sets the select, input_1 and input_2 to the given parameters, returns the output.

2.6) Data Memory

Keeps the data stored in memory. Based on a map and stores the address-data pair on the *dataMap*. Has control signals *memRead* and *memWrite*, without these signals are set, writing and reading operations can not be made.

To its constructor, a file path can be passed as a parameter. It reads addresses and its values from this file and adds these to the map. An example file format:



```
address value
28      4
19      45
3       6
```

void setMemRead(bool memRead):

Sets the *memRead* signal to the given parameter.

void setMemWrite(bool memWrite):

Sets the *memWrite* signal to the given parameter.

int read(int address):

Given *memRead* is true, returns the data on the *address*, if address has no value associated with it returns -1.

void write(int address, int data):

Given *memWrite* is true, adds the *address-data* pair to the *dataMap.*

3) Stalls

There are different cases with different number of stalls. If an instruction does not depend on any of the two instructions before it, a stall is not inserted unless a branch needs to be taken.
The stall cases and number of stalls inserted are:

load -> add/sub/or/and/sd (dependent on load)	=> 1 stall
load -> beq(dependent on load)	=> 3 stalls (if not taken 2 stalls)
add/sub/or/and -> beq(dependent on add)	=> 2 stalls (if not taken 1 stall)
beq	=> 1 stall (if not taken 0 stalls)

4) Addressing

PC is incremented by 1 to get the next instruction.

5) Main

All units are constructed in main.

5.1) Units

A *PC, registers, if_id, id_ex, ex_mem, mem_wb* are register files with sizes 1, 32, 6, 15, 32, 32, respectively.

There are *Instruction Memory, Alu, Data Memory* and *Control Unit* functions as explained above.

There is an *alu mux* that chooses the second input for Alu, a *pc mux* that chooses the next PC value when beq instruction comes, a *write mux* that selects the value to be written to register file at the end of the cycle.

There is an adder to perform add operation, a *branch alu* to compare two registers of the branch instruction by performing subtraction, a *branch address alu* to add PC and offset of the branch instruction.

5.2) Instruction Codes

0 -> add	1 -> sub	2 -> and	3 -> or
4 -> ld	5 -> sd	6 -> beq	7 -> nop

5.3) Pipeline

This is a pipelined processor with 5 stages: IF, ID, EX, MEM, WB. Their details can be found on the next section.

5.4) Execution in One Cycle

5.4.1) IF Stage

The instruction that PC points is read from Instruction Memory.

IF/ID Registers

IF/ID has 6 registers. They are set according to these indexes:

0 -> instruction code 1 -> regDest 2 -> reg1 3 -> reg2 4-> PC
5 -> offset

regDest, reg1, reg2 and offset are determined based on the instruction as below:

add || sub || and || or	<regDest>,<reg1>,<reg2>
ld	<regDest>, offset(<reg2>)
sd	<reg2>, offset(<reg1>)
beq	<reg1>,<reg2>, label
nop	regDest, reg1, reg2, offset are 0

5.4.2) ID Stage

The instruction in if_id is passed to the Control Unit. Control Unit determines this instruction's control signals.

- *Load-Use Hazard Detection*

  When id_ex has the ld instruction, id_ex's regDest and if_id's reg1 & reg2 are compared to detect dependency. If there is dependency, then *nop* instruction is inserted to stall the pipeline. Also if_id's registers and PC are set to its previous values. If there is no dependency, then the PC is incremented by 1.

  A load use hazard example:
  ```
  ld    x1, 3(x2)
  add   x3, x1, x3   (x1 is dependent to previous instruction)
  ```

- *Forwarding (MEM/WB -> ID/EX)*

  When mem_wb has the ld or arithmetic instruction, mem_wb's regDest and if_id's reg1 & reg2 are compared to see if there is a dependency. If so, it passes the required value to id_ex register. This passed value will be used at the next clock cycle in EX stage. This is forwarding the value to the instruction three cycle behind.

  An example forwarding of this type:
  ```
  ld    x1, 3(x2)
  add   x4, x1, x0  (1 cycle stall - load use hazard )
  sub   x3, x1, x0  (sub's x1 is dependent to ld's result)
  ```

Branch calculations are made in ID stage by adding extra units.

- *Branch*

  Branch After Load

  If there is a dependency between ld and beq, Load-Use Hazard will detect it and stall the pipeline one time. However beq uses its registers in ID stage, it should wait in ID until ld passes the updated value from MEM/WB. This requires one more stall.
  If ex_mem's instruction is ld and if_id's is beq, this section compares registers to check if there is a dependency. If so, it stalls the pipeline one more time.
  This type of stall example:

  ```
  ld      x1, 0(x2)
  beq     x1, x2, label
  ```

  (1 stall from load use hazard and 1 stall from branch after load, then ld can pass the x1's updated value from MEM/WB)

  Branch After Arithmetic Operation

  If there is a dependency between arithmetic operation and beq, beq should wait in ID stage until the arithmetic instruction updates its result in the EX/MEM register. This requires one stall.
  If id_ex's instruction is arithmetic and if_id's is beq, this section compares registers to detect if there is a dependency between instructions. If so, it stalls the pipeline one time.
  An example of this type of stall:

  ```
  add     x1, x2, x2
  beq     x1, x3, label
  ```

  (After 1 stall, add can pass x1's value from EX/MEM to IF/ID)

  Setting Branch Alu Inputs

  Before comparing two registers in branch alu to see if branch is taken, cases that require forwarding should be checked.
  EX/MEM and MEM/WB's regDest and branch's registers are compared to see if there is any dependency. If so, updated values of branch registers are passed from EX/MEM or MEM/WB and these values are set as branch alu inputs.

  Branch Result & Prediction

  If *branch alu* outputs zero, the branch is taken. if_id's registers and PC are set to the same values again to cancel the fetched instruction after branch. A nop instruction is inserted to stall the pipeline after a taken branch.
  If *branch alu* outputs other than zero, branch is not taken. There is no stall after branch. The next instruction after branch continues in pipeline. PC is incremented by 1 to fetch the second instruction after branch.
  In this flow when branch instruction is fetched, it is *predicted as not taken*. Next instruction is fetched at the next cycle.

ID/EX Registers

ID/EX has 15 registers. They are set according to these indexes:

0 -> instruction code	1 -> regDest	2 -> reg1	3 -> reg2	4-> PC

and they are taken from the ID/EX registers. Remaining registers contain:

5 -> value of reg1	6 ->value of reg2	7 -> regWrite
8 -> aluSrc	9 -> memRead	10-> memWrite

11 -> memToReg 12 -> pcSrc 13 -> aluOp
14 -> offset
Control signals are taken from IF/ID registers.

5.4.3) EX Stage

Address and data computations are made based on the control signals from ID/EX registers. Consists of an Alu Unit and a Mux.

Data Hazard Forwarding:

Data hazards are detected and corrected. If the inputs of the alu depend on the previous instruction, results are forwarded from EX/MEM registers, if depend on the instru**ction before the previous one** results are forwarded from MEM/WB registers. These forwarded values are given as inputs to Mux and Alu and the result is saved on EX/MEM registers. One case is the instruction after load. After a load if there is a dependency on the next instruction, a stall is inserted and the next instruction gets the value of the next instruction from MEM/WB registers.

Example of these cases:

```
add x5, x4, x3
add x6, x5, x5
```

Value of x5 forwarded from EX/MEM

```
add x5, x4, x3
add x6, x7, x7
add x1, x5, x4
```

Value of x5 forwarded from MEM/WB

```
ld x5, 13(x3)
add x1, x5, x6
```

Value of x5 forwarded from MEM/WB

EX/MEM Registers

EX/MEM has 15 registers. They are set according to these indexes:
0 -> instruction code 1 -> regDest 2 -> reg1 3 -> reg2 4-> PC
and they are taken from the ID/EX registers. Remaining registers contain:
5 -> Alu output 6 -> Alu Zero output 7 -> Adder output
8-> regWrite 9 -> aluSrc 10 -> memRead
11-> memWrite 12 -> memToReg 13 -> pcSrc
14 -> aluOp
Control signals are taken from ID/EX registers.

5.4.4) MEM Stage

Consists of the Data Memory Unit. Control signals memRead and memWrite are given to the Data Memory. The address calculated by the Alu is given as the data address, the reg2 value is given as the data value which is used only for "sd" instruction. Writing or reading is done depending on the memWrite and memRead signals.

MEM/WB Registers
EX/MEM has 10 registers. They are set according to these indexes:
0 -> instruction code 1 -> regDest 2 -> reg1 3 -> reg2 4-> PC
and they are taken from the IF/ID registers. Remaining registers contain:
5 -> Data read from the data memory for the address calculated by the Alu and saved on EX/MEM registers 5. index
6 -> Alu output from EX/MEM registers 5. index 7 -> regWrite
8-> memToReg 9 -> memRead
Control signals are taken from EX/MEM registers.

5.4.5) WB Stage

The regWrite signal is calculated on ID stage and forwarded by intermediate registers,destination register and the value that needs to be written to the register is given to the registers RegisterFile. The data that will be passed is determined by a Mux whose select is memToReg signal. If memToReg signal is set, data read from Data Memory (MEM/WB index 5) is passed to RegisterFile, otherwise the result of the Alu (MEM/WB index 6) is given.

5.5) Clock Cycle and Update of Registers

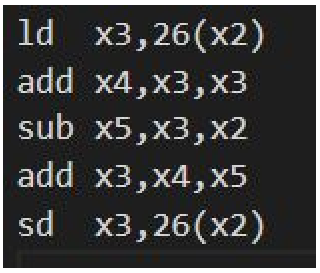
To replicate the behaviour of a real processor, register files are only updated when the update() function is called. Until then all written registers are stored in a temporary storage and can not be read from outside. When update() function is called, the temporary registers pass their values to the real registers and the old values are overwritten. Normally since all registers are updated on the rising clock edge, the value should not change during the clock cycle and the same value should be read and this condition is also satisfied on this project. All the units that are based on RegisterFile class (Intermediate registers, Register File, PC) are updated at the end of the clock cycle.

Without stalls, a program takes instruction number+4 clock cycles which is consistent with a 5 stage pipeline. If any of the stall conditions occur, this number increases.

6) Example Runs

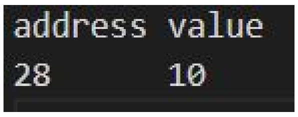
6.1) Example 1

Input:



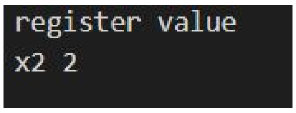
```
ld  x3,26(x2)
add x4,x3,x3
sub x5,x3,x2
add x3,x4,x5
sd  x3,26(x2)
```

Memory:



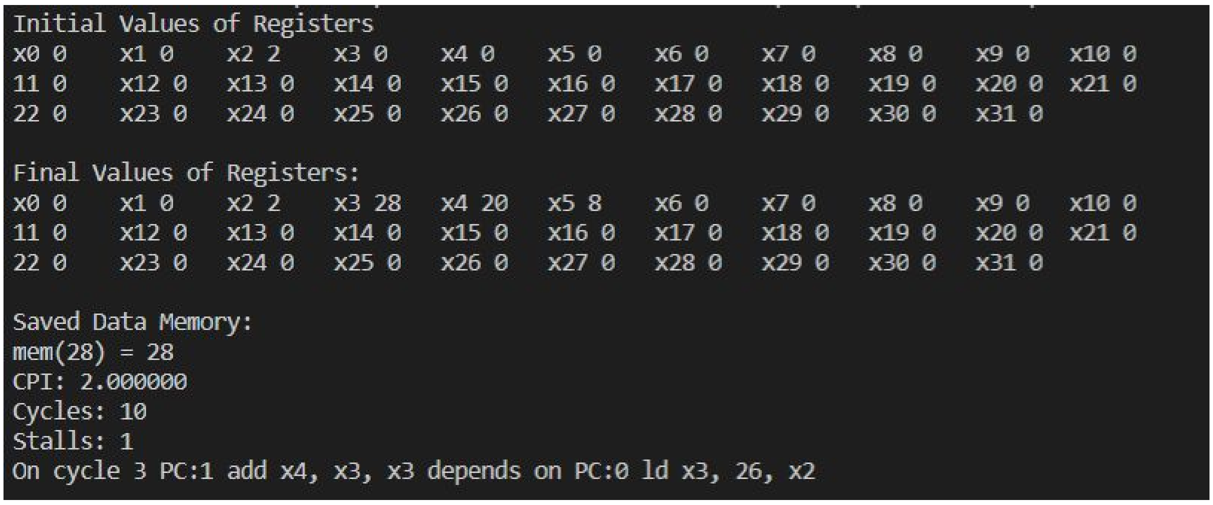
```
address value
28       10
```

Registers:



```
register value
x2 2
```

Output:



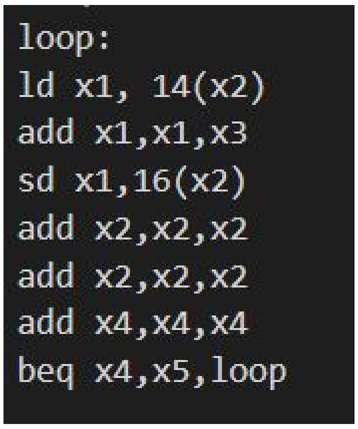
```
Initial Values of Registers
x0 0    x1 0    x2 2    x3 0    x4 0    x5 0    x6 0    x7 0    x8 0    x9 0    x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0   x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Final Values of Registers:
x0 0    x1 0    x2 2    x3 28   x4 20   x5 8    x6 0    x7 0    x8 0    x9 0    x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0   x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Saved Data Memory:
mem(28) = 28
CPI: 2.000000
Cycles: 10
Stalls: 1
On cycle 3 PC:1 add x4, x3, x3 depends on PC:0 ld x3, 26, x2
```

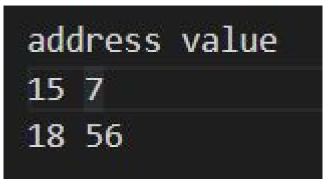
6.2) Example 2

Input:



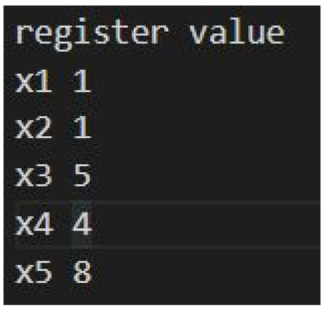
```
loop:
ld x1, 14(x2)
add x1,x1,x3
sd x1,16(x2)
add x2,x2,x2
add x2,x2,x2
add x4,x4,x4
beq x4,x5,loop
```

Memory:



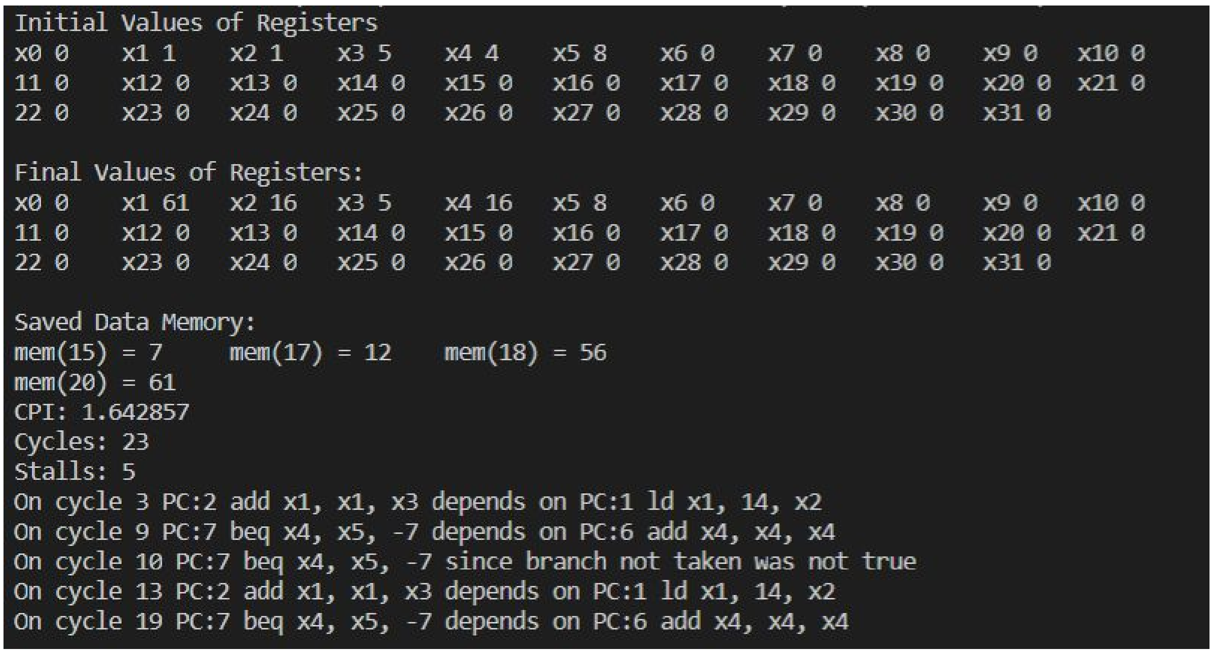
```
address value
15 7
18 56
```

Registers:



```
register value
x1 1
x2 1
x3 5
x4 4
x5 8
```

Output:



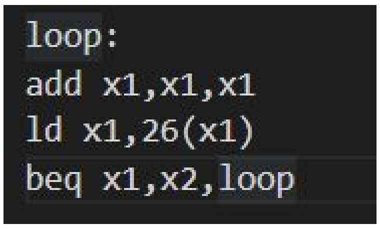
```
Initial Values of Registers
x0 0    x1 1    x2 1    x3 5    x4 4    x5 8    x6 0    x7 0    x8 0    x9 0   x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0  x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Final Values of Registers:
x0 0    x1 61   x2 16   x3 5    x4 16   x5 8    x6 0    x7 0    x8 0    x9 0   x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0  x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Saved Data Memory:
mem(15) = 7     mem(17) = 12    mem(18) = 56
mem(20) = 61
CPI: 1.642857
Cycles: 23
Stalls: 5
On cycle 3 PC:2 add x1, x1, x3 depends on PC:1 ld x1, 14, x2
On cycle 9 PC:7 beq x4, x5, -7 depends on PC:6 add x4, x4, x4
On cycle 10 PC:7 beq x4, x5, -7 since branch not taken was not true
On cycle 13 PC:2 add x1, x1, x3 depends on PC:1 ld x1, 14, x2
On cycle 19 PC:7 beq x4, x5, -7 depends on PC:6 add x4, x4, x4
```

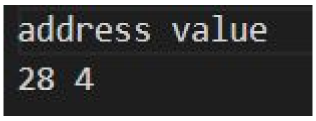
6.3) Example 3

Input:



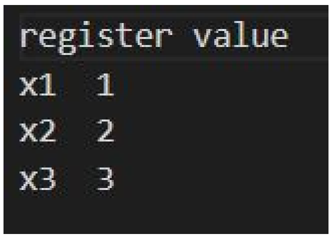
```
loop:
add x1,x1,x1
ld x1,26(x1)
beq x1,x2,loop
```

Memory:



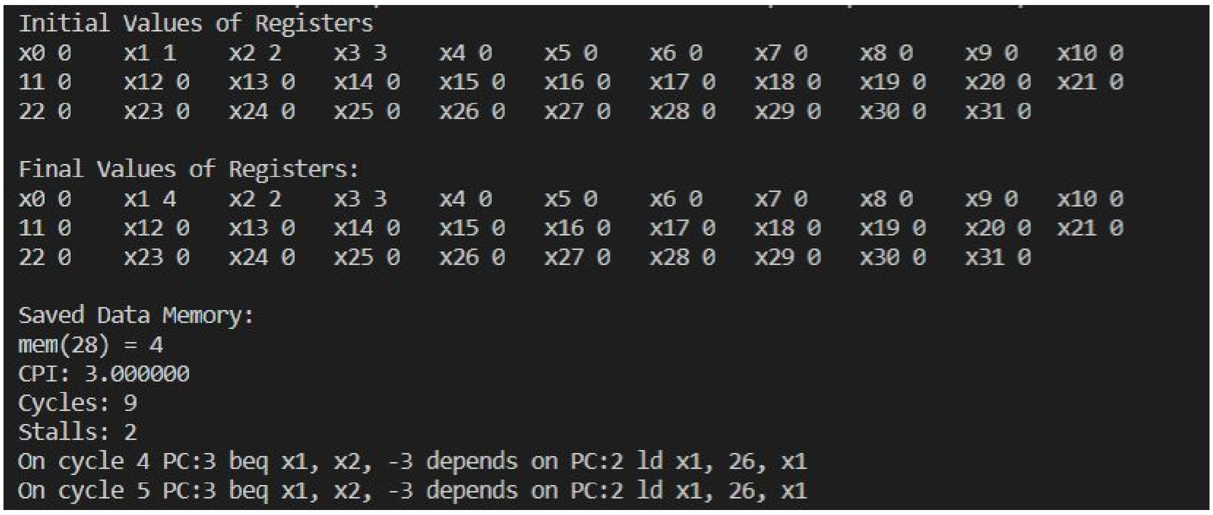
```
address value
28 4
```

Registers:



```
register value
x1   1
x2   2
x3   3
```

Output:



```
Initial Values of Registers
x0 0    x1 1    x2 2    x3 3    x4 0    x5 0    x6 0    x7 0    x8 0    x9 0   x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0  x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Final Values of Registers:
x0 0    x1 4    x2 2    x3 3    x4 0    x5 0    x6 0    x7 0    x8 0    x9 0   x10 0
11 0    x12 0   x13 0   x14 0   x15 0   x16 0   x17 0   x18 0   x19 0   x20 0  x21 0
22 0    x23 0   x24 0   x25 0   x26 0   x27 0   x28 0   x29 0   x30 0   x31 0

Saved Data Memory:
mem(28) = 4
CPI: 3.000000
Cycles: 9
Stalls: 2
On cycle 4 PC:3 beq x1, x2, -3 depends on PC:2 ld x1, 26, x1
On cycle 5 PC:3 beq x1, x2, -3 depends on PC:2 ld x1, 26, x1
```