Design of a pipelined RV32I processor

Ashwin Rajesh (Sr. No 21519)
M.Tech Microelectronics and VLSI Design
Indian Institute of Science, Bangalore

***Abstract*— This report summarises the design of a pipelined RISC-V processor as part of the Processor System Design course by Prof. Kuruvilla Varghese at the Indian Institute of Science, Bengaluru.**

I. Problem statement

Design and Implement a 5-stage Pipelined RISC-V processor on Digilent BASYS3 FPGA Board. Target Device is Xilinx Artix-7 XC7A35T-ICPG236C (Family Artix-7, Part XC7A35T, Package CPG236, Speed Grade -1). The Processor is 32-bit RISC-V Processor. The processor should support basic arithmetic-logic instructions, branch instructions, load-store instructions, and jump instructions. Implement hazard detection and forwarding (from EX, MEM and WB stage outputs). Implement the Stall unit for load, and branch instructions. You can assume aligned transferfor word, half-word and byte data. For data and instruction memory, you can use the memory IP made of Block RAM of FPGA. Memory size can be 1Kword. Test the processor with a suitable algorithm, with a procedure call.

II. Instruction set architecture

The processor supports the base instruction set of the 32 bit RISC-V ISA, called RV32I for short except the ECALL, EBREAK, FENCE and CSR instructions. 37 instructions are supported, belonging to one of R, I, S, B, U or J types. Roughly, they can be divided into Load/Store, R type and I type arithmetic/logical, and branch/jump instructions.

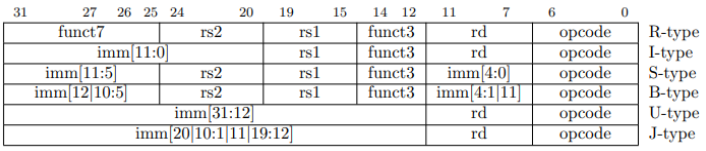


Fig. 2: Instruction Formats

III. Datapath

The pipeline consists of 5 stages : Fetch, Decode, Execute, Memory and Writeback. Apart from the datapath, it also has stall logic to handle some data and control hazards. An abstract diagram of the datapath is shown in Fig. 1. A detailed diagram is given in Appendix A.

A. Fetch stage

The fetch unit fetches the instructions from the instruction memory.

It consists of the instruction memory and next Program Counter (PC) logic. The next program counter value is decided based on the current PC value in the FE/DE pipeline register and the branch target and status in the EX/MEM pipeline register.

The instruction memory is a BRAM with synchronous read, hence its input address is the next program counter value. The instruction is directly fed to the decode logic because the instruction memory effectively has a builtin output register because of its synchronous read.

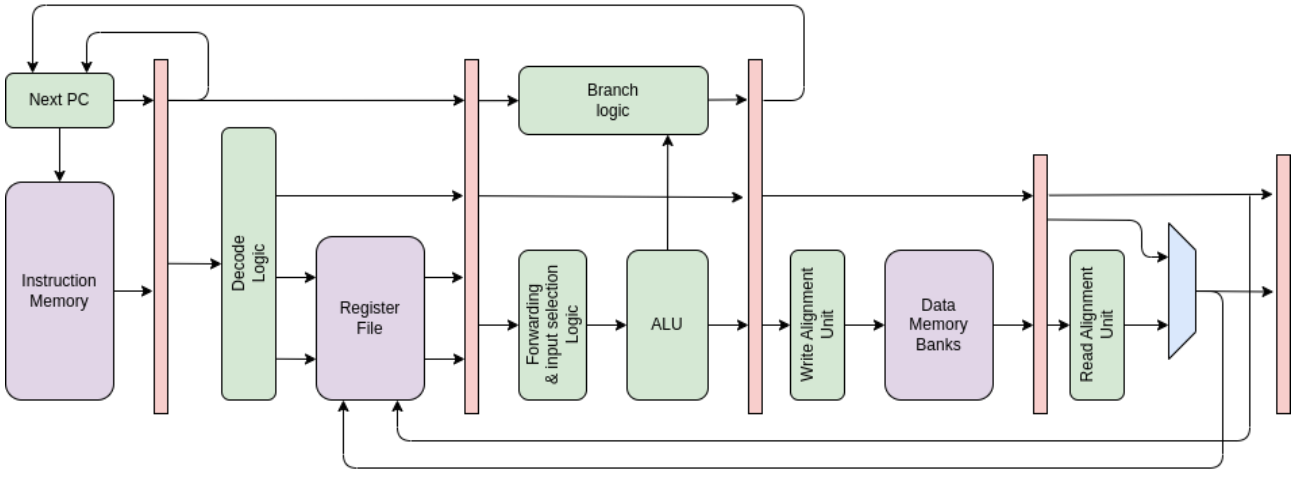


Fig. 1 : Abstract RV32I pipeline diagram

It also maintains a valid bit. This is used for adding bubbles in the pipeline. Since the instruction register is from the BRAM itself, it cannot be zeroed out for inserting a bubble. Hence, we use another bit which will be used by other logic including the decode stage and stall stage for ignoring that instruction.

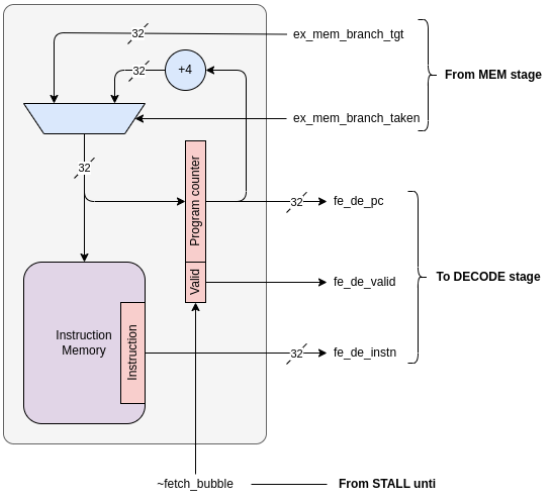


Fig. 3: Fetch stage

B. Decode stage

The decode stage decodes the instruction into appropriate control signals for the ALU, branch logic, memory and register file and reads data from register file and immediate fields in the instruction.

It is designed as multiple separate decode units, each responsible for a part of the decoding process. The instruction decoder extracts the opcode, func7, func3 and format of the instruction. Further, the immediate decoder decodes the immediate part for each instruction format. The register decoder decodes the rs1, rs2 and rd addresses and register write enable signal based on the opcode, instruction and func7 and func3 fields. The **register write signal is zeroed out for writes to R0** by the register decoder to simplify forward and stall logic.

There are 3 decoders, one each for ALU, branch and memory control signals. The ALU control decoder decodes the ALU compute and evaluation modes and the inputs and output selects for the ALU inputs and output. The branch control decoder decodes if the instruction is a branch instruction, if its a conditional branch and if the base of the branch target is from RS2 or PC.

For the stall logic, if a bubble is to be inserted in decode stage, or if the fetched instruction was invalid (fe_de_valid is 0), the register write enable, memory read and write enable and branch enable signals are zeroes out to ensure that the instruction does not have any impact on the state of subsequent instructions through register/memory writes, forwarding, stalling or branches.

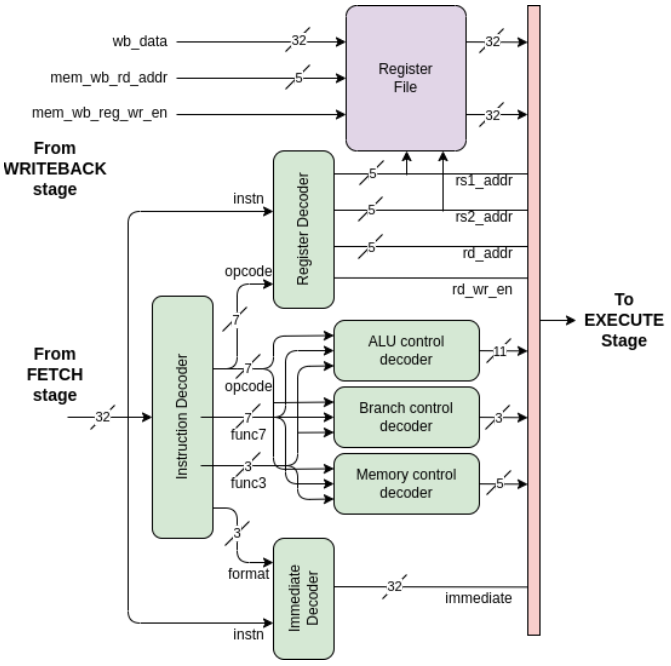


Fig. 4: Decode stage

C. Execute stage

The execute stage computes the result to be stored back into a register or the address of a memory access, and computes the branch status and target for branch and jump instructions. It consists of an ALU that computes the result of arithmetic and logical operations and evaluates conditions like <, ==, >= for signed and unsigned operations, a forwarding unit, branch compute unit and an adder for computing the branch target address.

The values for the input 1 and input 2 ALU source control signals for inputs are tabulated below. They are fed to multiplexers which choose the ALU inputs.

Select value	Source	Used for
IN1_RS1	Forwarded RS1	Default
IN1_0	32’b0	Passing through in2 (lui)
IN1_PC	Program Counter	Write next PC JALR, JAL
IN2_RS2	Forwarded RS2	Default
IN2_4	32’b4	Write next PC JALR, JAL
IN2_IMM	Immediate	Immediate instructions

Table 1: ALU source control signals

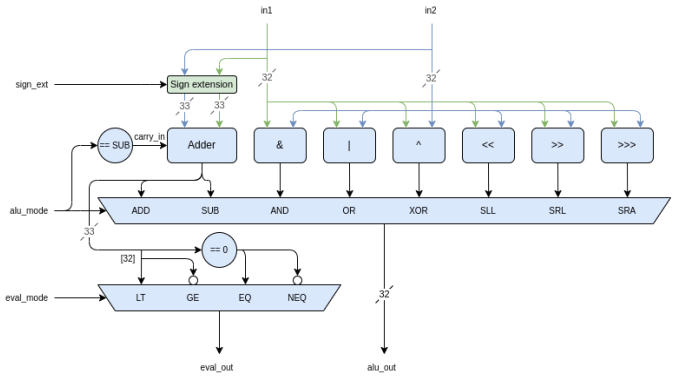


Fig. 5: Arithmetic and Logical Unit

The ALU mode is used to choose the output operation, ALU evaluation mode is used to choose what comparison is used to generate the final evaluate output and sign extend bit chooses whether the inputs must be sign extended or zero-padded to 33 bits before addition. This is useful for unsigned comparison operations where 0 padding is required. For signed comparisons, sign extension is performed. This allows the same compare operations to be performed for signed and unsigned comparison instructions by just changing this bit. This design shares the adder for ADD, SUB and all the signed and unsigned comparison (including SLT, SLTI) instructions.

The ALU output bit is used to select the zero-padded eval_out value to the pipeline register for the SLT and SLTI instructions.

The branch unit uses the decoded branch control signals to multiplex different signals to form the branch target and branch status as can be seen in Fig. 6. The branch target is always the sum of the immediate field and a “branch base” which can be either the PC or forwarded RS1 value.

For inserting a bubble in this stage, the same procedure as decode bubble is followed, i.e zeroing out the memory read/write, register write and branch status pipeline registers to prevent any state updates.

D. MEMORY STAGE

The mem stage handles reads and writes to/from data memory. The data memory is a synchronous read/write BRAM. Because of the synchronous reads, the outputs are directly fed to the next stage.

To enable reads and writes to bytes and half-words, a read and write alignment circuit is required along with four BRAM memory banks. The read alignment is done in the writeback stage because the read values are available only in the next cycle. The write alignment circuit generates the write enable and write data values for each bank depending on the write address and memory write mode (byte, half-word or word). Unaligned access checks are not made in hardware so errors can be caused which will go undetected!

The stall signal is also fed into the BRAM as part of its read enable to prevent a stalled instruction in writeback stage losing its memory read value.

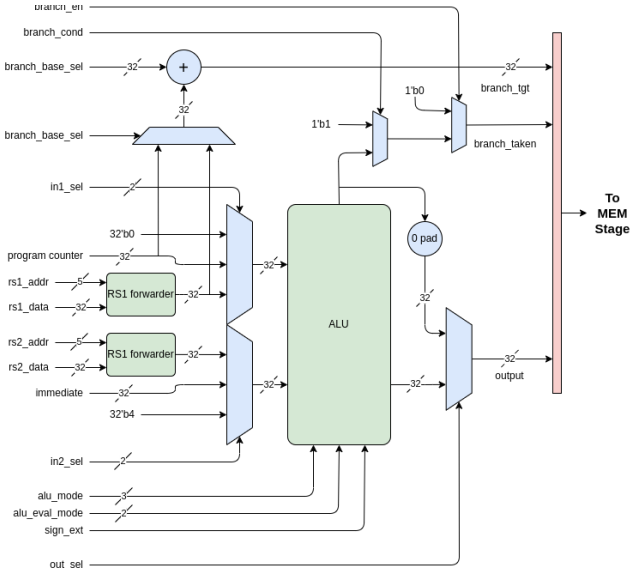


Fig. 6: Execute stage

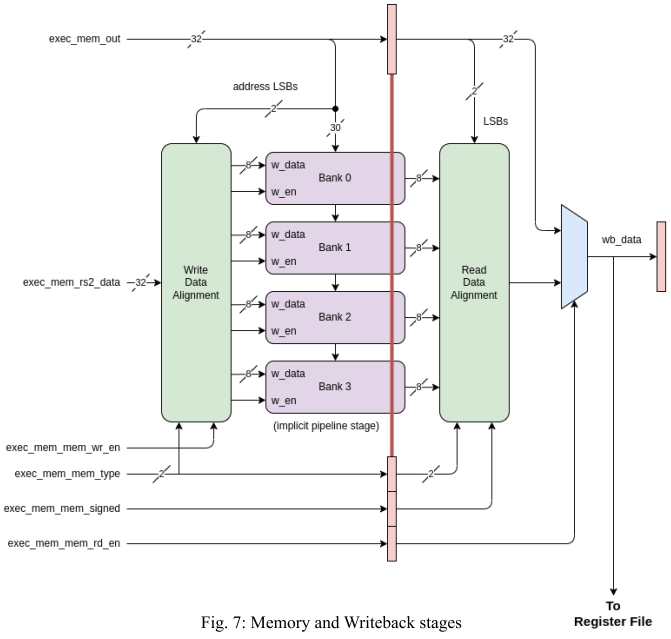


Fig. 7: Memory and Writeback stages

E. Writeback stage

In the writeback stage, the memory read data alignment is performed based on the memory access mode (byte, half-word or word and signed/unsigned) and then the final write value is decided based on the memory read enable bit. If this bit is set, register rd is written from memory read signal, else the register is written from the ALU output value which was passed through the MEM stage.

IV. Hazard detection and mitigation

There are RAW data hazards and control hazards in a typical five-stage pipleine. For handling most data hazards, a forwarding unit is used in the execute stage.

The forward unit compares the RS1/RS2 address in the EX stage and RD address in the EX/MEM, MEM/WB and WB pipeline registers. If they match and the corresponding register write enable signal is set, the data is forwarded from the corresponding stage. Writes to R0 which should not be forwarded, are handled simply by setting write enable to 0 when rd is 0 is the decode stage.

This works for forwarding results from any instruction that generates its output in the EX stage. However, load instructions which generate data in the MEM stage (and alignment in WB stage) will not be able to forward data to any instruction that may immediately succeed it, which would be in the EX stage. This necessisates inserting a bubble in the EX stage and stalling the rest of the pipeline. A stall unit is used which detects such conditions and generates appropriate stall and bubble signals.

Another condition necessitating adding bubbles are control hazards caused by any branch or jump instruction. The next PC value in such instructions will only be generated after the EX stage and till then we cannot fetch any new instructions. Hence, we insert bubbles in the fetch stage by setting the valid signal to 0 in the fetch stage. The stall unit ensures that when a bubble is added in a stage, all preceding stages are stalled. The stall unit easily recognizes this condition using the branch enable signal that was decoded for branch logic control.

To stall a stage of the pipeline, the registers simply retain the old value. To insert a bubble, in the fetch stage, the valid bit is set to 0 and in DE and EX stages, the register write, memory read and write and branch enable signals are set to 0. An invalid instruction in decode stage also leads to a bubble being inserted (no register or memory writes or branches). Stalls take precedence over bubbles to prevent instructions being lost while stalled. The processor has a single global stall signal input which will stall all stages in the processor. This can be used like an enable signal to pause the processor operation.

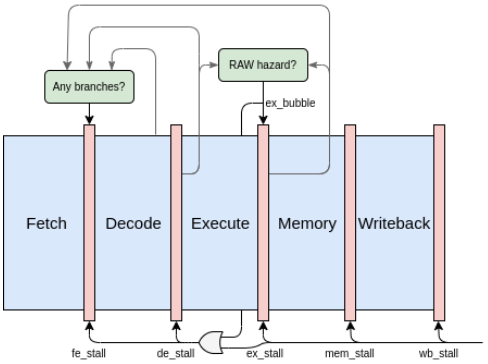


Fig. 8: Stall and bubble generation logic

V. Results

The design was implemented and tested on the basys3 board for generating numbers from the fibonacci sequence at a maximum clock frequency of 100MHz. The worst negative slack was 0.117ns giving a maximum frequency of **101.183MHz**. About 9.27% of LUTs and 5% of BRAM resources were used for implementing the design.

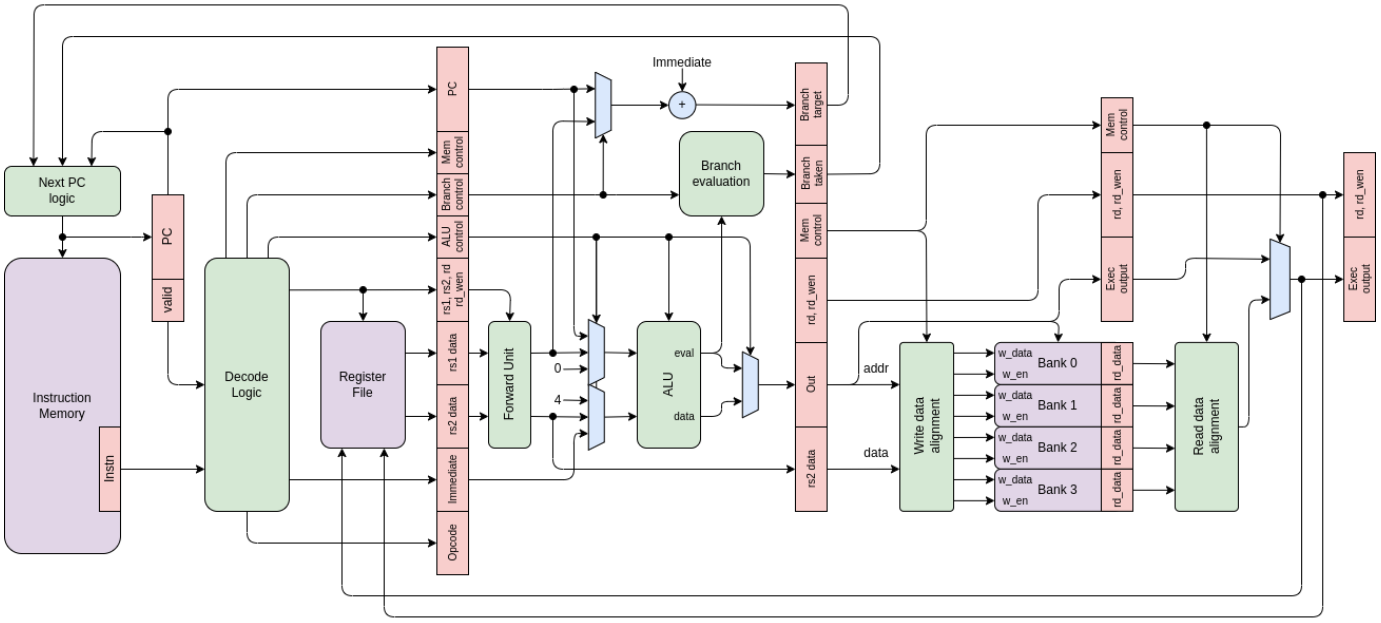
VI. References

[1] The RISC-V Instruction Set Manua, *Andrew Waterman, SiFive inc*

[2] Digital Design and Computer Architecture, RISC-V Edition, *Sarah L Harris, David Money Harris*

[3] Computer Organization and Design “The Hardware/Software Interface: RISC-V Edition” *David A. Patterson, John L. Hennessy*

APPENDIX



Pipelined RISC-V datapath

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.117 ns	Worst Hold Slack (WHS):	0.055 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4097	Total Number of Endpoints:	4097	Total Number of Endpoints:	1454

All user specified timing constraints are met.

Timing report summary

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	Slice (8150)	LUT as Logic (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N toplevel	1929	1443	256	694	1929	2.5	14	1
btnc_dbnc (debouncer)	10	21	0	12	10	0	0	0
btnu_dbnc (debouncer_0)	9	21	0	11	9	0	0	0
proc_inst (rv32i)	1861	1384	256	668	1861	2.5	0	0
alu (alu)	123	0	0	84	123	0	0	0
data_mem_b0 (data_memory)	14	0	0	14	14	0.5	0	0
data_mem_b1 (data_memory_1)	497	0	0	194	497	0.5	0	0
data_mem_b2 (data_memory_2)	77	0	0	48	77	0.5	0	0
data_mem_b3 (data_memory_3)	111	0	0	72	111	0.5	0	0
inst_mem (inst_memory)	92	0	0	40	92	0.5	0	0
reg_file (regfile)	906	992	256	458	906	0	0	0
seven_seg_out_inst (seven_segment_out)	49	17	0	20	49	0	0	0

Utilization report summary

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	`lb rd, imm(rs1)`	load byte	`rd = SignExt([Address]7:0)`
0000011 (3)	001	–	I	`lh rd, imm(rs1)`	load half	`rd = SignExt([Address]15:0)`
0000011 (3)	010	–	I	`lw rd, imm(rs1)`	load word	`rd = [Address]31:0`
0000011 (3)	100	–	I	`lbu rd, imm(rs1)`	load byte unsigned	`rd = ZeroExt([Address]7:0)`
0000011 (3)	101	–	I	`lhu rd, imm(rs1)`	load half unsigned	`rd = ZeroExt([Address]15:0)`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010111 (23)	–	–	U	`auipc rd, upimm`	add upper immediate to PC	`rd = {upimm, 12'b0} + PC`
0100011 (35)	000	–	S	`sb rs2, imm(rs1)`	store byte	`[Address]7:0 = rs27:0`
0100011 (35)	001	–	S	`sh rs2, imm(rs1)`	store half	`[Address]15:0 = rs215:0`
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	`[Address]31:0 = rs2`
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs24:0`
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >> rs24:0`
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>> rs24:0`
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110111 (55)	–	–	U	`lui rd, upimm`	load upper immediate	`rd = {upimm, 12'b0}`
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

Supported Instructions

```c
void delay();
register int i asm ("s11");
register int d asm ("s10");

asm ("addi sp, s0, 1024");

void delay(int);

int main() {
    int i_prev = 0;
    i = 1;
    d = 10000000;
    while(1) {
         delay(d);
         i      = i + i_prev;
         i_prev = i - i_prev;
     }
}

void delay(int delay) {
     for(int i = 0; i < delay; i++);
}
```

C code used for testing the processor (fibonacci)

```asm
addi sp, s0, 1024

main:
  addi sp,sp,-32
  sw ra,28(sp)
  sw s0,24(sp)
  addi s0,sp,32
  sw zero,-20(s0)
  li s11,1
  li a5,9998336
  addi s10,a5,1664
.L2:
  mv a5,s10
  mv a0,a5
  call _Z5delayi
  mv a4,s11
  lw a5,-20(s0)
  add a5,a4,a5
  mv s11,a5
  mv a4,s11
  lw a5,-20(s0)
  sub a5,a4,a5
  sw a5,-20(s0)
  j .L2

_Z5delayi:
  addi sp,sp,-48
  sw s0,44(sp)
  addi s0,sp,48
  sw a0,-36(s0)
  sw zero,-20(s0)
.L5:
  lw a4,-20(s0)
  lw a5,-36(s0)
  bge a4,a5,.L6
  lw a5,-20(s0)
  addi a5,a5,1
  sw a5,-20(s0)
  j .L5
.L6:
  nop
  lw s0,44(sp)
  addi sp,sp,48
  jr ra
```

Assembly code used
(generated with gcc, has some pseudo instructions)