**RV32I Base Instruction Set**

This datasheet is written to help my design of RV32I base instruction set processor based on systemc.

Different sections are divided by different instructions in the instruction set.

**Section 1: ALU instructions**

ALU instructions are the simplest instructions in the whore instruction set. There are several operations in this kind of instructions.

1. The alu operation. There are 10 instructions in this instruction list.
2. Arithmetic operations: ADD and SUB.
3. Logical operations: XOR, OR and AND.
4. Set less than: SLT and SLTU. The set less than operations places the value 1 in register rd if register rs1 < rs2 when both are treated as signed or unsigned numbers, else 0 is written to rd.
5. Shift operations: SLL, SRL and SRA. The first letter stand for “shift” the second letter stand for shift - “left” or “right”, and the third letter stand for the shift style – “logical” or “arithmetic”. To explain the difference between logical and arithmetic shift, I will take right shift operation for example. When a number is right shifted logically, the right most bit will be dumped, and 0 will be added to the left most bit. In the operation of arithmetic shift, on the other hand, the left most bit will be assigned to the bit of the original left most bit value.

All the shift operations shift on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2. For all the other seven instructions, the two operands are the output of rs1 and rs2.

For this part of the instructions, a 32 bit ALU is needed and a control signal that is capable of differing the ten different operations (at least 4 bits) is needed for the ALU. Because there are both signed and unsigned operations so 6 signals (actually they should not be called signals) are needed to be declared, two signed and two unsigned 32-bit integers and two unsigned 5-bit integers (shift amount). **PS:** I’m still kind of confuse about who to write the registers between different stages, so this ALU will just be used for these 10 instructions, and the output will be sent to next stage sequentially, the other arithmetic operations needed by other instructions will be implemented by other ALUs. ALU will have a 32-bit sequential output, obviously. **PS:** for STL and SLTU instruction 1 or 0 will be assigned to the 32-bit output.

1. The second part of ALU instructions is to store the data to rd. So a 1-bit register-write signal should come from the previous stage (control stage) should go pass ALU stage sequentially and go pass the DRAM stage sequentially and then go backward to register stage. So the register-write signal will go through 3 registers. Similar to this signal, a 5-bit rd address which is in the 11-7 bit of the instructions, will go through the same stages and then backward to register stage.

In summary for ALU instructions:

1. One 32-bit ALU just for ALU instructions. **(alu\_main)**
2. 1-bit register-write signal generate by register stage, decided by the opcode (if ALU instructions, then 1), go through three layers of registers and then backward to the input of registers. **(alu\_main\_reg\_w)**
3. 5-bit rd address which is in the 11-7 bit of instruction bit, generated by register stage, just pick from the instruction bit, go through three layers of registers and then backward to the input of registers. **(alu\_main\_rd)**
4. 32-bit rs1, rs2 which is the output of the registers, generated by registers, go from control stage to ALU stage. **(rs1, rs2)**
5. 32-bit result generated by ALU stage, go through 2 layers of registers, then go backward to the input of registers. **(alu\_main\_result)**
6. 5-bit control signal generated by register stage, decided by opcode, funct3 and bit30 of the instruction, go through 1 layer of register to ALU stage to decide ALU operation. **(alu\_main\_opcode)**

**Section 2: ALUI instructions**

The ALUI instructions are slightly different from ALU instructions. The operands of ALUI instructions are rs1 and immediate (bit 31-20).

1. The ALUI operation. There are 10 instructions in this instruction list.
2. Arithmetic operation: ADDI.
3. Logical operations: XORI, ORI and ANDI. Operands are rs1 and the **signed extended** of immediate.
4. Set less than: SLTI and SLTIU. The set less than operations places the value 1 in register rd if register rs1 < signed or unsigned extended of immediate, else 0 is written to rd.
5. Shift operations: SLLI, SRLI and SRAI. The basic operation is same as ALU shifts. The shift amount is bit 24-20 of the instruction bits. Bit 30 of the instruction indicate whether the shift is logical or arithmetic (1 stands for arithmetic, 0 for logical).

ALUI instructions could all be the same as ALU instructions, except for two extra components. The 1st one is the signed and unsigned extended units which is easy to handle in systemc, just declare signed and unsigned 12-bit integer and assign the immediate to them. The tricky part is the shift immediate, which needed to be passed directly from last stage as an input, and then signed to 5-bit unsinged integer.

1. Store the data back to register.

This part of ALUI is exactly the same as ALU

Summary for ALUI instructions:

1. ALUI can share the ALU unit with ALU instructions, a 32-bit ALU. **(alu\_main)**
2. **(alu\_main\_reg\_w)**
3. **(alu\_main\_rd)**
4. **(rs1, rs2)**
5. **(alu\_main\_result)**
6. Because ALUI will share the ALU unit, I think the alu opcode should be changed to 5-bit. **(alu\_main\_opcode)**
7. 12-bit immediate taken from bit 31-20 of instruction, go through 1 layer of register from register stage to ALU stage. **(alu\_imm)**
8. 5-bit shift amount, taken from 24-20 bit of instruction, go through 1 layer of register from register stage to ALU stage. **(shift\_imm)**

**Section 3: JAL instruction**

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. (in this version 4 bytes, which is 32 btis) The offset is sign-extended and added to the pc to form the jump target address. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register and x5 as an alternate link register.

If we just consider JAL, it is able to generate the jump address in the current stage (control stage), but JALR and BRANCH instructions both generate the destination address in the ALU stage, so just for the convenience, we let JAL also generate jump address in the ALU stage.

There two operations in the JAL instruction:

1. Store the data of pc + 4 into rd. To achieve this, a 12-bit unsigned adder is needed in the control stage, and after the result need to be unsigned extended to 32 bit and store in address rd. A register write signal also need to be generated by the opcode in control stage.
2. Of course, you need to deal with the jumping part. Because this processor use 12 bit address so the jumping length will be {12, 20, 30-21} bits of the instruction. The jump length will be added to pc of current instruction in the ALU stage, and then backward to the input of pc. So pc will also be passed from control stage to ALU stage. Also a jump flag will be generated by opcode in control stage, go through the register layer to the ALU stage and then backward to the input of PC.

Summary for JAL instruction:

1. 12-bit pc of the current instruction, generated by pc passing through 2 layers of register to the ALU stage, pc will be added to jump length, and the result will go backward to the input of PC. **(pc\_jb)**
2. A 12-bit adder in ALU stage to add pc to signed jump length, and the result go backward to the input of PC through 0 layer of register. **(alu\_jb)**
3. 12-bit jump length will be generated in control stage from the instruction bits, and go through 1 layer of register and go to alu\_jb as a signed 12-bit integer. **(jump\_length)**
4. 1-bit flag to store the data pc + 4 to rd will be generated in control stage by opcode, and feed to the registers. **(store\_pc)**
5. A 12-bit adder will be also be placed in control stage to calculate pc + 4, and also the result will be unsigned extended to 32 bit and stored to rd. **(alu\_pc4)**
6. 1-bit flag to make pc jump to the jump address will be generated in control stage by opcode, and then go through 1 layer of register and feed to the input of pc. **(pc\_jump)**

**Section 4: JALR instruction**

JALR is similar to JAL instruction, but instead of adding pc to the immediate, this JALR add the data stored in register rs1 to the immediate, and use it as the jump address. All the other operations are the same as JAL.

There are two parts of JAL instruction:

1. Store the data of pc + 4 into rd. To achieve this, a 12-bit unsigned adder is needed in the control stage, and after the result need to be unsigned extended to 32 bit and store in address rd. A register write signal also need to be generated by the opcode in control stage.
2. Of course, you need to deal with the jumping part. Because this processor use 12 bit address so the jumping length will be {31-20} bits of the instruction. The jump length will be added to rs1 in the ALU stage, and then backward to the input of pc. Also a jump flag will be generated by opcode in control stage, go through the register layer to the ALU stage and then backward to the input of PC.

Summary for JAL instruction:

1. 12-bit pc of the current instruction, generated by pc passing through 1 layers of register to the control stage, pc will be added 4 and stored in rd. **(pc\_jb)**
2. A 12-bit adder in ALU stage to add immediate to rs1, and the result go backward to the input of PC through 0 layer of register. **(alu\_jb)**
3. A 1-bit flag is needed for alu\_jb to select weather to add pc or rs1 to the immediate, pc 0, rs1 1. **(jump\_select)**
4. 12-bit jump length will be generated in control stage from the instruction bits, and go through 1 layer of register and go to alu\_jb as a signed 12-bit integer. The value of jump\_length will be decided by opcode, because JAL and JALR are different. **(jump\_length)**
5. 1-bit flag to store the data pc + 4 to rd will be generated in control stage by opcode, and feed to the registers. **(store\_pc)**
6. A 12-bit adder will be also be placed in control stage to calculate pc + 4, and also the result will be unsigned extended to 32 bit and stored to rd. **(alu\_pc4)**
7. 1-bit flag to make pc jump to the jump address will be generated in control stage by opcode, and then go through 1 layer of register and feed to the input of pc. **(pc\_jump)**

**Section 5: BRANCH instructions**

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets, and is added to the current pc to give the target address.

BEQ, branch if rs1 and rs2 are equal; BNE, branch if rs1 and rs2 are not equal;

BLT, branch if rs1 is less than rs2. BLTU, the unsigned version of BLT;

BGE: branch if rs1 is greater or equal to rs2; BGEU: unsigned version of BGE;

BRANCH instructions can be divided in three parts:

1. Read the **rs1 and rs2**, and pass **pc** to ALU stage, generate a **branch flag**, also pass the **funct3 bits** to ALU stage for different operations of BRANCH instructions.
2. Compare the rs1 and rs2, in order to do this, **a 32 bit alu is need in ALU stage**. Also funct3 is used to decide whether to branch.
3. **A flag of branch instruction ins\_branch** will be passed to ALU stage, and after the comparison by branch alu, **a branch flag** will be generated for pc to decide whether to branch.

Summary for BRANCH instructions:

1. 1-bit branch flag will be generated by opcode in control stage, the flag will go through 1 layer of register to ALU stage, help decide whether pc should branch. **(ins\_branch)**
2. A 32-bit alu will be generated in ALU stage to compare rs1 and rs2. **(alu\_if\_branch)**
3. 3-bit funct3 which is bit 14-12 of the instruction will be generated in the control stage and go through 1 layer of register passed to ALU stage to help decide if pc should branch. **(funct3)**
4. 12-bit pc will be passed from PC stage through 1 layer of register to control stage in order to be added to the immediate. **(pc\_jb**)
5. 12-bit immediate will be generate in the control stage by {31, 11, 30-25, 10-7} bits of instruction and added to pc\_jb in the control stage. **(branch\_imm)**
6. A 12-bit adder will be generated in control stage to add pc\_jb to branch\_imm. **(alu\_branch)**
7. 12-bit output of branch length will be generated by alu\_branch in control stage and go through 1 layer of register to ALU stage and then go backward to the input of pc. **(branch\_length)**

**Section 6: LOAD instructions**

LOAD instructions load 32-bit data from registers to DRAM, there are two operations in the instruction:

1. Add rs1 and the immediate to get the data address of DRAM in ALU stage.
2. Store the data in dram to register rd.
3. There are 5 different LOAD instructions, so after reading data from DRAM the data need to be extended in different methods in control stage.

Summary for LOAD instructions:

1. 1-bit load flag generated in control stage by opcode go through 3 layers of register backward to the input of register to write in the register. **(load\_w)**
2. A load-extended unit in control stage, input is funct3, load flag, rd, and 32-bit data from DRAM. **(load\_extended)**
3. 32-bit DRAM output generated by DRAM feed to load\_extended. **(dram\_out)**
4. 3-bit funct3 generated in control stage go through 3 layers of register backward to control stage. **(funct3)**
5. 5-bit rd address generated in control stage go through 3 layers of register backward to control stage **(rd)**
6. 12-bit immediate generated in control stage go through 1 layer of register feed to alu\_ls to calculate the address of DRAM. **(immediate)**
7. A load store ALU unit in ALU stage to add rs1 and immediate, the 12-bit result will be feed to DRAM. **(alu\_ls)**
8. 12-bit DRAM address, generated in ALU stage by alu\_ls go through 1 layer of register to DRAM. **(dram\_addr)**

Section 7: STORE instructions

STORE instructions store the data in registers to DRAM, 3 STORE instructions. There are three operations in STORE instructions:

1. Calculate the DRAM address
2. Store the data to DRAM.
3. Extended to different data to DRAM.

Summary for STORE instructions:

1. 1-bit store flag generated in control stage by opcode go through 2 layers of register backward to the input of DRAM to write in the DRAM. **(store\_w)**
2. A store-extended unit in ALU stage, input is funct3, rs2. **(store\_extended)**
3. 32-bit rs2 feed to store\_extended. **(rs2)**
4. 3-bit funct3 generated in control stage go through 3 layers of register backward to control stage. **(funct3)**
5. 12-bit immediate {31-25, 11-7} generated in control stage go through 1 layer of register feed to alu\_ls to calculate the address of DRAM. **(immediate)**
6. A load store ALU unit in ALU stage to add rs1 and immediate, the 12-bit result will be feed to DRAM. **(alu\_ls)**
7. 12-bit DRAM address, generated in ALU stage by alu\_ls go through 1 layer of register to DRAM. **(dram\_addr)**

Important!!

The register need to be able to be written and read at the same time, for JALR and maybe some other instructions.