

# ISA for the CS 3220 Project

Opcodes are 4 bits.

Functions (secondary opcodes) are 4 bits.

Immediate operands are 16 bits.

Register indices are 4 bits (RS1, RS2, RD).

Rough Verilog example:

```
wire[31:0] iword;
wire[3:0] rd, rs1, rs2
wire[15:0] imm;
wire[3:0] fn;
wire[3:0] opcode;

assign opcode = iword[3:0];
assign fn = iword[7:4];
assign imm = iword[23:8];
assign rs2 = iword[23:20];
assign rs1 = iword[27:24];
assign rd1 = iword[31:28];
```

## ALU-R

```
ADD    : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 0000 0000"}
SUB    : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 0001 0000"}
AND    : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 0100 0000"}
OR     : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 0101 0000"}
XOR    : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 0110 0000"}
NAND   : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 1100 0000"}
NOR    : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 1101 0000"}
XNOR   : {fmt: "RD,RS1,RS2",iword: "RD RS1 RS2 000000000000 1110 0000"}
```

## ALU-I

```
ADDI   : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 0000 1000"}
SUBI   : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 0001 1000"}
ANDI   : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 0100 1000"}
ORI    : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 0101 1000"}
XORI   : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 0110 1000"}
NANDI  : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 1100 1000"}
NORI   : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 1101 1000"}
XNORI  : {fmt: "RD,RS1,imm",iword: "RD RS1 imm[15:0] 1110 1000"}
MVHI   : {fmt: "RD,imm", iword: "RD 0000 imm[15:0] 1011 1000"}
```

## Load/Store

```
LW     : {fmt: "RD,imm(RS1)", iword: "RD RS1 imm[15:0] 0000 1001"}
SW     : {fmt: "RS2,imm(RS1)",iword: "RS1 RS2 imm[15:0] 0000 0101"}
```

## CMP-R

```
F      : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 0000 0010"}
EQ     : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 0001 0010"}
LT     : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 0010 0010"}
LTE    : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 0011 0010"}
T      : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 1000 0010"}
NE     : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 1001 0010"}
GTE    : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 1010 0010"}
GT     : {fmt: "RD,RS1,RS2", iword: "RD RS1 RS2 000000000000 1011 0010"}
```

## CMP-I

```
FI     : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 0000 1010"}
EQI    : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 0001 1010"}
LTI    : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 0010 1010"}
LTEI   : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 0011 1010"}
TI     : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 1000 1010"}
NEI    : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 1001 1010"}
GTEI   : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 1010 1010"}
GTI    : {fmt: "RD,RS1,imm", iword: "RD RS1 imm[15:0] 1011 1010"}
```

## BRANCH

```
BF     : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 0000 0110"}
BEQ    : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 0001 0110"}
BLT    : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 0010 0110"}
BLTE   : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 0011 0110"}
BEQZ   : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 0101 0110"}
BLTZ   : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 0110 0110"}
BLTEZ  : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 0111 0110"}

BT     : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 1000 0110"}
BNE    : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 1001 0110"}
BGTE   : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 1010 0110"}
BGT    : {fmt: "RS1,RS2,imm", iword: "RS1 RS2 imm[15:0] 1011 0110"}
BNEZ   : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 1101 0110"}
BGTEZ  : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 1110 0110"}
BGTZ   : {fmt: "RS1,imm", iword: "RS1 0000 imm[15:0] 1111 0110"}

JAL    : {fmt: "RD,imm(RS1)", iword: "RD RS1 imm[15:0] 0000 1011"}
```

## PSEUDO INSTRS

B is implemented using BEQ

```
BR      : {fmt: "imm", itext: ["BEQ R6,R6,imm"]}
```

NOT is implemented using NAND

```
NOT     : {fmt: "RD,RS", itext: ["NAND RD,RS,RS"]}
```

BLE, BGE are implemented using LTE/GTE and BNEZ

```
BLE     : {fmt: "RS1,RS2,imm", itext: ["LTE R6,RS1,RS2","BNEZ R6,imm"]}
```

```
BGE     : {fmt: "RS1,RS2,imm", itext: ["GTE R6,RS1,RS2","BNEZ R6,imm"]}
```

CALL/RET/JMP are implemented using JAL

```
CALL    : {fmt: "imm(RS1)", itext: ["JAL RA,imm(RS1)"]}
```

```
RET     : {fmt: "", itext: ["JAL R9,0(RA)"]}
```

```
JMP     : {fmt: "imm(RS1)", itext: ["JAL R9,imm(RS1)"]}
```

## Single cycle processor specification

Requirements for the processor:

- It must implement the documented ISA.
- PC must start at (byte address) 0x40.
- SW to address 0xF0000000 must display bits 15 to 0 as hexadecimal digits on the HEX display.
- SW to address 0xF0000004 must display bits 9 to 0 on LEDR.
- SW to address 0xF0000008 must display bits 7 to 0 on LEDG.
- LW from address 0xF0000010 must read the current KEY state. The result should be 0 when no KEY pressed, and 0xF when all are pressed.
- LW from address 0xF0000014 reads SW state.
- The 32-bit value we read should really be {22'b0,SWd) where SWd is a debounced value of SW.

## Assembler specification

The assembler must read an assembly file containing a program that follows the ISA spec, and it must output a MIF file with 2048 32-bit words of memory (8192 bytes in total).

### Opcode and function mappings

For the following tables, the rows indices represent the most significant bits (MSB) and the column indices are the LSB.

#### General opcode mapping

LSB MSB	00	01	10	11
00	ALU-R		CMP-R	
01		SW	BRANCH	
10	ALU-I	LW	CMP-I	JAL
11				

#### ALU-R/ALU-I function mapping

LSB MSB	00	01	10	11
00	ADD/ADDI	SUB/SUBI		
01	AND/ANDI	OR/ORI	XOR/XORI	
10				
11	NAND/NANDI	NOR/NORI	NXOR/NXORI	

### CMP-R/CMP-I function mapping

MSB \ LSB	00	01	10	11
00	F/FI	EQ/EQI	LT/LTI	LTE/LTEI
01				
10	T/TI	NE/NEI	GTE/GTEI	GT/GTI
11				

### BRANCH function mapping

MSB \ LSB	00	01	10	11
00	BF	BEQ	BLT	BLTE
01		BEQZ	BLTZ	BLTEZ
10	BT	BNE	BGTE	BGT
11		BNEZ	BGTEZ	BGTZ

### Instruction format

- ALU-R
  - $rd = rs1 \text{ op } rs2$
- CMP-R
  - $rd = (rs1 \text{ op } rs2) ? 1 : 0$
- Store
  - $Mem[rs1 + \text{signextension}(imm)] = rs2$
- Load
  - $rd = Mem[rs1 + \text{signextension}(imm)]$
- ALU-I
  - $rd = rs1 \text{ op } \text{signextension}(imm)$
- CMP-I
  - $rd = (rs1 \text{ op } \text{signextension}(imm)) ? 1 : 0$
- BRANCH
  - if  $(rs1 \text{ op } rs2)$   $PC = PC + 4 + (\text{signextension}(imm) * 4)$
- JAL
  - $rd = PC + 4$
  - $PC = rs1 + 4 * \text{signextension}(imm)$

### Assembler syntax

- Instruction opcodes and register names:

- Are **reserved words (can't be used as labels).**
  - Appear in **either lowercase or uppercase.**
  - If there is a destination register, it is listed first.
- Labels:
  - Are created using a name and then ":" at the start of a line
  - **Corresponds to the address where label created**
- Immediate operands – number or label
  - If number, hex (C format, e.g. 0xffff) or decimal (can have - sign)
  - If label, just use the name of the label (without ":")
    - For PC-relative, the **immediate field is label\_addr-PC-4**
    - For other insts, the **immediate field is 16 least-significant bits of label\_addr**
- Each register has multiple names:
  - R0..R3 are also A0..A3 (function arguments, caller saved)
  - R3 is also RV (return value, caller saved)
  - R4..R5 are also T0..T1 (temporaries, caller saved)
  - R6..R8 are also S0..S2 (callee-saved values)
  - R9 reserved for assembler use
  - R10..R11 reserved for system use (we'll see later for what)
  - R12 is GP (global pointer)
  - R13 is FP (frame pointer)
  - R14 is SP (stack pointer)
    - Stack grows down, SP points to lowest in-use address
  - R15 is RA (return address)

### Special assembler instructions

- .ORIG <number>
  - Changes "current" address to <number>
- .WORD <value>
  - Places 32-bit word <value> at the current address
  - <value> **can be a number or a label name**
  - If label name, value is the full 32-bit label\_addr
- .NAME <name>=<value>
  - Defines a name (label) with a given value (number)