Instruction Format for 32 bit CPU

We have a total of 33 instructions supported by CPU each of which can be encoded into 6 bit opcodes each and keep the remaining 26 bits for operands and immediate values. But while doing so, we will lose readability of the instruction set. So to make it easy to read as well as instruct, we would be splitting our 32-bit instruction into several parts.

Partition on the basis of instruction sets:

1. Arithmetic and Logical Instructions
2. Arithmetic and Logical instructions (Immediate values)
3. Logical Not operation
4. Load-Store Instructions
5. Branch Instructions
6. Stack Instructions
7. Register to register transfer
8. Program Control Instructions

Since as a whole there were 6 instruction sets, the encoding would have 3 bits for 2^3=8 possibilities. There are 18 kinds of operations possible in the ALU instruction set which would have taken 5 bits to encode with a lot of waste options.. So, to conserve the bits for taking larger immediate values, we use the remaining 8-6=2 possibilities of the encoding scheme in the first partition to take care of those extra ALU operations.

Register addressing

We have 16 general purpose registers along with one special purpose register, i.e., stack pointer (SP) which can be directly accessed by the programmer. So, in total 17 registers need addressing for a programmer to utilize them. Hence, each register is addressed by a 5 bit encoding. Apart from that we also have a program control (PC) register which is not accessed directly.

Registers R0-R15 are represented by normal counting, like, R5 = 00101, whereas SP is addressed as 11111.
Now, the following is our whole instruction set for the 32-bit CPU.

ARITHMETIC AND LOGICAL INSTRUCTIONS						
Operation	**Opcode**	**Function**	**Filler bits**	**Reg Src 2**	**Reg Src 1**	**Reg Dest**
ADD	001	000	9’b0	xxxxx	xxxxx	xxxxx

SUB	001	001	9'b0	xxxxx	xxxxx	xxxxx
AND	001	010	9'b0	xxxxx	xxxxx	xxxxx
OR	001	011	9'b0	xxxxx	xxxxx	xxxxx
XOR	001	100	9'b0	xxxxx	xxxxx	xxxxx
SLA	001	101	9'b0	00000	xxxxx	xxxxx
SRA	001	110	9'b0	00000	xxxxx	xxxxx
SRL	001	111	9'b0	00000	xxxxx	xxxxx
ARITHMETIC AND LOGICAL INSTRUCTIONS (IMMEDIATE)						
Operation	**Opcode**	**Function**	**Filler bits**	**Imm16**	**Reg Src 1**	**Reg Dest**
ADDI	010	000	-	16 bits	xxxxx	xxxxx
SUBI	010	001	-	16 bits	xxxxx	xxxxx
ANDI	010	010	-	16 bits	xxxxx	xxxxx
ORI	010	011	-	16 bits	xxxxx	xxxxx
XORI	010	100	-	16 bits	xxxxx	xxxxx
SLAI	010	101	-	16 bits	00000	xxxxx
SRAI	010	110	-	16 bits	00000	xxxxx
SRLI	010	111	-	16 bits	00000	xxxxx
LOGICAL NOT OPERATION						
Operation	**Opcode**	**Function**	**Filler bits**	**Imm16**	**Reg Src**	**Reg Dest**
NOT	011	000	16'b0	-	xxxxx	xxxxx
NOTI	011	001	-	16 bits	00000	xxxxx
LOAD-STORE INSTRUCTION						
Operation	**Opcode**	**Function**	**Filler bits**	**Imm16**	**Reg Src 1**	**Reg Target**
LD	100	00	1'b0	16 bits	xxxxx	xxxxx

ST	100	11	1'b0	16 bits	xxxxx	xxxxx
LDSP	100	01	1'b0	16 btis	xxxxx	11111
STSP	100	10	1'b0	16 bits	xxxxx	11111
BRANCH INSTRUCTIONS						
Operation	**Opcode**	**Function**	**Filler bits**	**Imm22**		**Reg Src**
BR	101	11	-	22 bits		xxxxx
BMI	101	01	-	22 bits		xxxxx
BPL	101	10	-	22 bits		xxxxx
BZ	101	00	-	22 bits		xxxxx
STACK INSTRUCTIONS						
Operation	**Opcode**	**Function**	**Filler bits**	**Imm22**		**Reg Target**
PUSH	110	10	22'b0	-		xxxxx
POP	110	01	22'b0	-		xxxxx
CALL	110	11	-	22 bits		00000
RET	110	00	22'b0	-		00000
REGISTER TO REGISTER TRANSFER						
Operation	**Opcode**		**Filler bits**		**Reg Src**	**Reg Dest**
MOVE	111		19'b0		xxxxx	xxxxx
PROGRAM CONTROL INSTRUCTIONS						
Operation	**Opcode**	**Function**	**Filler bits**			
HALT	000	0	28'b0			
NOP	000	1	28'b0			

Data Path for the 32-bit Processor

The data path can be mainly broken down into five parts, *IF*, *ID*, *EX*, *MEM*, and *WB*. The following is a brief description of each part.

1. Instruction Fetch (IF):

The control unit generates the address for the instruction in memory, which is the current *PC* value. The instruction is fetched from the instruction memory, and stored in the Instruction Register (*IR*). The *PC* is then incremented by 4 using an adder, and its value is stored in the *NPC* register.

2. Instruction Decode (ID):

The opcode and other necessary fields are extracted from the instruction in the Instruction Register (*IR*). The control unit determines which registers in the Register Bank, it also sends the control signals to do the necessary operations' padding and sign extension to the immediate value. Consequently, the values of the registers are read from the Register File and stored in the *A* and *B* registers while the immediate values are stored in the *IMM16* and *IMM22* registers.

A multiplexer is used to select which of the immediate values to use (if they are used) which gets the control signal from the control unit.

If the stack pointer is involved in the instruction, the value of the stack pointer is calculated by either adding or subtracting 4 from it depending on the type of the instruction (push or pop). The value of the stack pointer is stored in the *NSP* register which is then written back to the stack pointer.

3. Execute (EX):

The *SelA/NPC'* control signal with the multiplexer determines which value to use as the first operand of the ALU. The *SelB/IMM'* control signal similarly determines which value to use as the second operand of the ALU. This is determined by the nature of the instruction. (R-type, I-type, J-type, etc.)

The ALU performs the necessary operation on the two operands determined by the control signal *ALUfunc*. The result of the ALU operation is stored in the *Z* register.

Meanwhile, the condition checker gets the control signal from the control unit and checks the type of branching instruction. Along with this the value of the register *A* is also evaluated. If the condition

is true with respect to the value of the register *A*, the branch is taken and the value of the *NPC* is updated with the value of the register *Z*. Otherwise, the branch is not taken and the value of the *NPC* is kept the same. Note that in case of unconditional branching, the branch is always taken i.e. the next signal sent to the multiplexer is always 1.

Note that the stack pointer also may change the value of *PC* but it will never overlap with a branching instruction. It will be described in the next section.

4. Memory (MEM):

In this stage, load and store instructions access data from the data memory.

Here, first in the case of branching, the condition value is checked. If the condition is true, the branch is taken and the value of the *NPC* is updated with the value of the register *Z*. Otherwise, the branch is not taken and the value of the *NPC* is kept the same. After this, the *PC* is updated with the value of the *NPC*.

Otherwise, if it's a load instruction, the value of the register *Z* is used as the address to access the data memory. The data read from the memory is stored in the register *LMD*. If it's a store instruction, the value of the register *Z* is used as the address to access the data memory. The data to be stored is taken from the register *B*.

In case of push and pop instructions, the value of the stack pointer is used as the address to access the data memory. The data read from the memory is stored in the register *LMD*. The value of the register *B* is used as the data to be stored in the memory. In case of call and return instructions, the value of the stack pointer is used to modify the value of the *PC*. The value of the stack pointer is also modified accordingly as was stored in the *NSP* register in the *ID* stage.

5. Write-Back (WB):

A multiplexer is used to select which value to write back to the register in the Register Bank. In case of Load instructions, the value of the register *LMD* is written back to the register in the Register Bank while in case of other instructions, the value of the register *Z* is written back to the register in the Register Bank.

List of Control Signals

1. PCin
2. NPCin

3. ReadIM
4. IRin
5. ReadRegPort1
6. ReadRegPort2
7. Ain
8. Bin
9. Imm16in
10. Imm22in
11. ImmSel
12. SelA/NPC’
13. SelB/Imm’
14. ALUfunc
15. SelCond
16. SPin
17. NSPin
18. Add/Sub’
19. Zin
20. MuxPC
21. SelMemSP
22. MuxWB
23. WriteReg
24. ReadMem
25. WriteMem
26. LMDin