DEPARTMENT OF ELECTRONIC & TELECOMMUNICATION ENGINEERING
UNIVERSITY OF MORATUWA

EN2031 - Fundamentals of Computer Organization and Design

Processor Design Project

Group Xeon

Name	Index No.
Gunawardana W. N. M	210199D
Maduwantha L.H.H	200370K
Dilshan N.L.	210129P

October 26, 2023

(a)(i) Instruction Set Architecture

(1) ALU Instructions:

1. ADD Rd, R1, R2: (Rd $\longleftarrow$ R1 + R2)
   Add the values in registers R1 and R2 and store the result in Rd.
2. SUB Rd, Rs, R2: (Rd $\longleftarrow$ R1 - R2)
   Subtract the value in register R2 from R1 and store the result in Rd.
3. AND Rd, R1, R2: (Rd $\longleftarrow$ R1 OR R2)
   Bitwise AND operation between R1 and R2, store the result in Rd.
4. OR Rd, R1, R2: (Rd $\longleftarrow$ R1 AND R2)
   Bitwise OR operation between R1 and R2, store the result in Rd.

All the ALU instructions are 3 operand type, and we use register direct addressing mode. But since the output is also stored in the accumulator after an ALU operation, having 2 registers is sufficient (i.e., Rd=R1).

(2) Data Transfer Instructions:

5. MOV Rd, Rs: (Rd $\longleftarrow$ Rs)
   Copy the value from register Rs to Rd.

(3) Stack Operations Instructions:

6. PUSH Rs:
   Push the value from register Rs onto the stack, decrement SP.
7. POP Rd:
   Pop the value from the stack into register Rd, increment SP.

(4) Memory Operations Instructions:

8. LOAD Rd, Ra: (Rd $\longleftarrow$ M[Ra])
   Load the value from Data Memory at the address specified by register Ra into Rd.
9. LOAD Rd, Ra!: (Rd $\longleftarrow$ M[Ra]) & (Ra $\longleftarrow$ Ra+1)
   Load the value from Data Memory at the address specified by Ra into Rd and increment Ra.
10. STORE Rs, Ra: (M[Ra] $\longleftarrow$ Rs)
    Store the value from Rs into Data Memory at the address specified by register Ra.
11. STORE Rs, Ra! (M[Ra] $\longleftarrow$ Rs) & (Ra $\longleftarrow$ Ra+1)
    Store the value from Rs into Data Memory at the address specified by Ra and increment Ra.

(5) Load Immediate Instructions:

12. LDI Rd, imm8: (Rd $\longleftarrow$ imm8)
    Load an 8-bit immediate value sign extended to 16 bits into Rd.
    There are two ways to select the destination bits of the immediate value in the destination register:
    - Rd $\longleftarrow$ imm[7:0] (Select the first 8 bits in Rd)

- Rd $\longleftarrow$ imm[15:8] (Select the second 8 bits in Rd)

Note: Here we use the little-endian byte ordering (like in RISC-V)

(6) Unconditional Branching Instructions:

13. JMP offset: (PC $\longleftarrow$ PC + offset)
    Branch to PC + offset.
    Offset is mostly an 8-bit immediate (imm[7:0])

(7) Conditional Branching Instructions:

First, let us see how to evaluate the value of SREG. For that, we should be given 2 registers (Ra & Rb):

- If Ra=Rb; SREG = 0
- If Ra>Rb; SREG > 0
- If Ra<Rb; SREG < 0

14. JEQ offset:
    Branch to PC + offset if the zero flag in SREG is set.
    Offset is mostly an 8-bit immediate (imm[7:0])
15. JGT offset:
    Branch to PC + offset if the greater-than flag in SREG is set.
    Offset is mostly an 8-bit immediate (imm[7:0])
16. JLT offset:
    Branch to PC + offset if the less-than flag in SREG is set.
    Offset is mostly a 8-bit immediate (imm[7:0])

(a)(ii) Instruction Format

In our system, we rely on a 16-bit architecture for our instructions. As we only have 16 instructions, we require 4 bits for opcodes, though we will use 5 bits. Additionally, we have 8 registers, with PC serving as a special case. To address registers within each instruction, we use 3 bits.

1st Format:

15	10	7	4 0
	R2	Rd/R1	Opcode

We can use this format for all the **ALU instructions.** Although we have specified 3 registers in ALU operations, usually the output is stored in the accumulator (we can use one of the input registers). So, 2 registers are sufficient.

Similarly, we can use this format for **Memory operation instructions** and **Data transfer instructions.** The reason is here also we need only the opcode and 2 register addresses.

2nd Format:

15	9	7	4	0
imm		Rd/R1	Opcode	

We can use this format for **Load Immediate Instructions, Unconditional Branching Instructions and Conditional Branching Instructions.** The reason is that, for all these instructions we need an opcode, one destination register and an 8-bit immediate value.

3rd Format:

15	7	4	0
	Rd/R1	Opcode	

Since the **pop instruction** only needs one destination address and opcode, we can use this format for it.

4th Format:

15	10	7	4	0
	R2		Opcode	

Since the **push instruction** only needs one source address and opcode, we can use this format for it. Since we use this position for source register addressing, we cannot use the 3rd instruction format for this.

(b) Instruction Encoding

We can just use binary codes to encode instructions. Although we can represent all the opcodes just by 4-bits, we will be using 5-bits for future flexibility.

(1) ALU Instructions:

- Add: 00000

- SUB: 00001
- AND: 00010
- OR: 00011

(2) Data Transfer Instructions:

- MOV: 00100

(3) Stack Operations Instructions:

- PUSH: 00101
- POP: 00110

(4) Memory Operations Instructions:

- LOAD: 00111
- LOAD!: 01000
- STORE: 01001
- STORE!: 01010

(5) Load Immediate Instructions:

- LDI: 01011

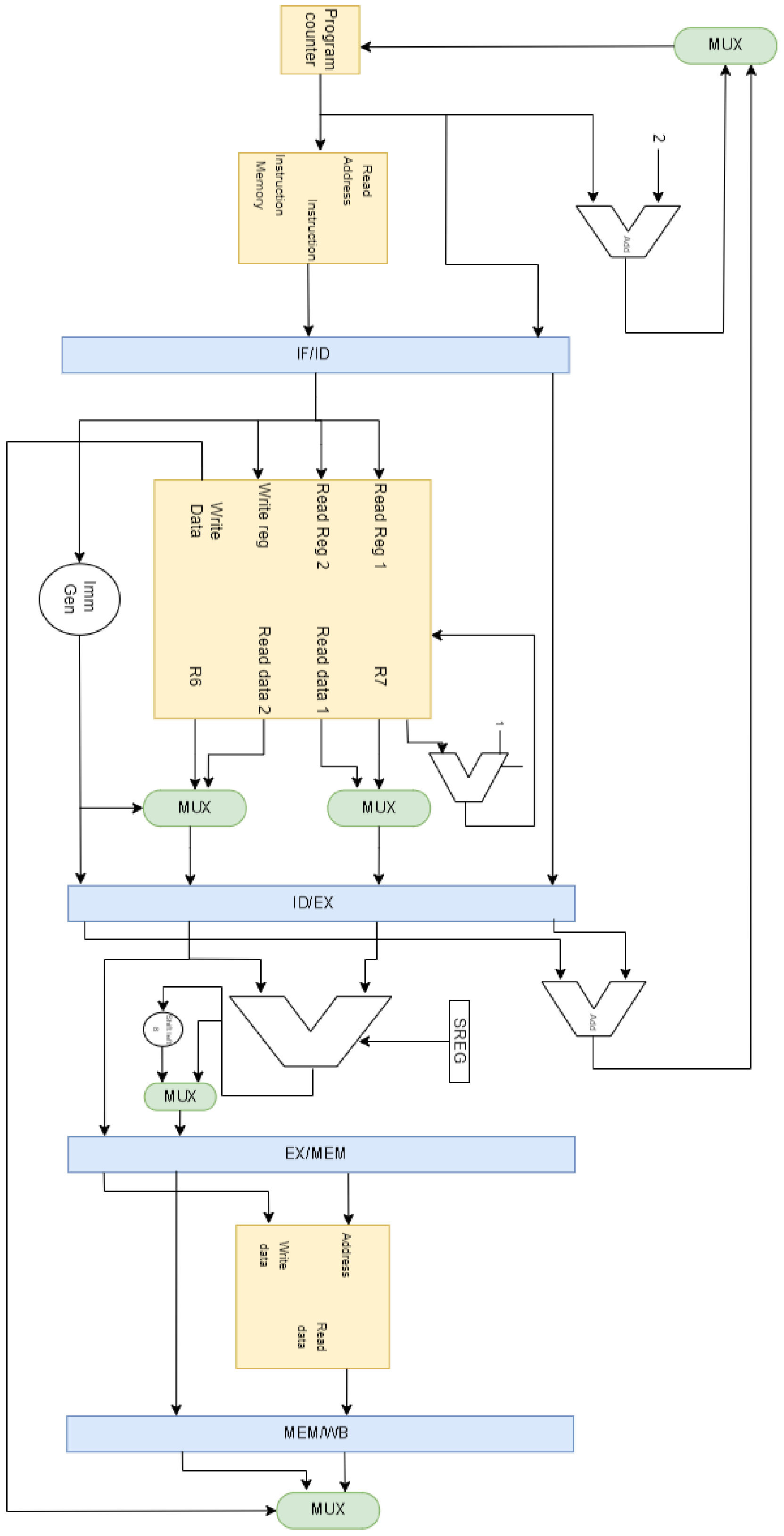
(6) Unconditional Branching Instructions:

- JMP: 01100

(7) Conditional Branching Instructions:

- JEQ: 01101
- JGT: 01110
- JLT: 01111

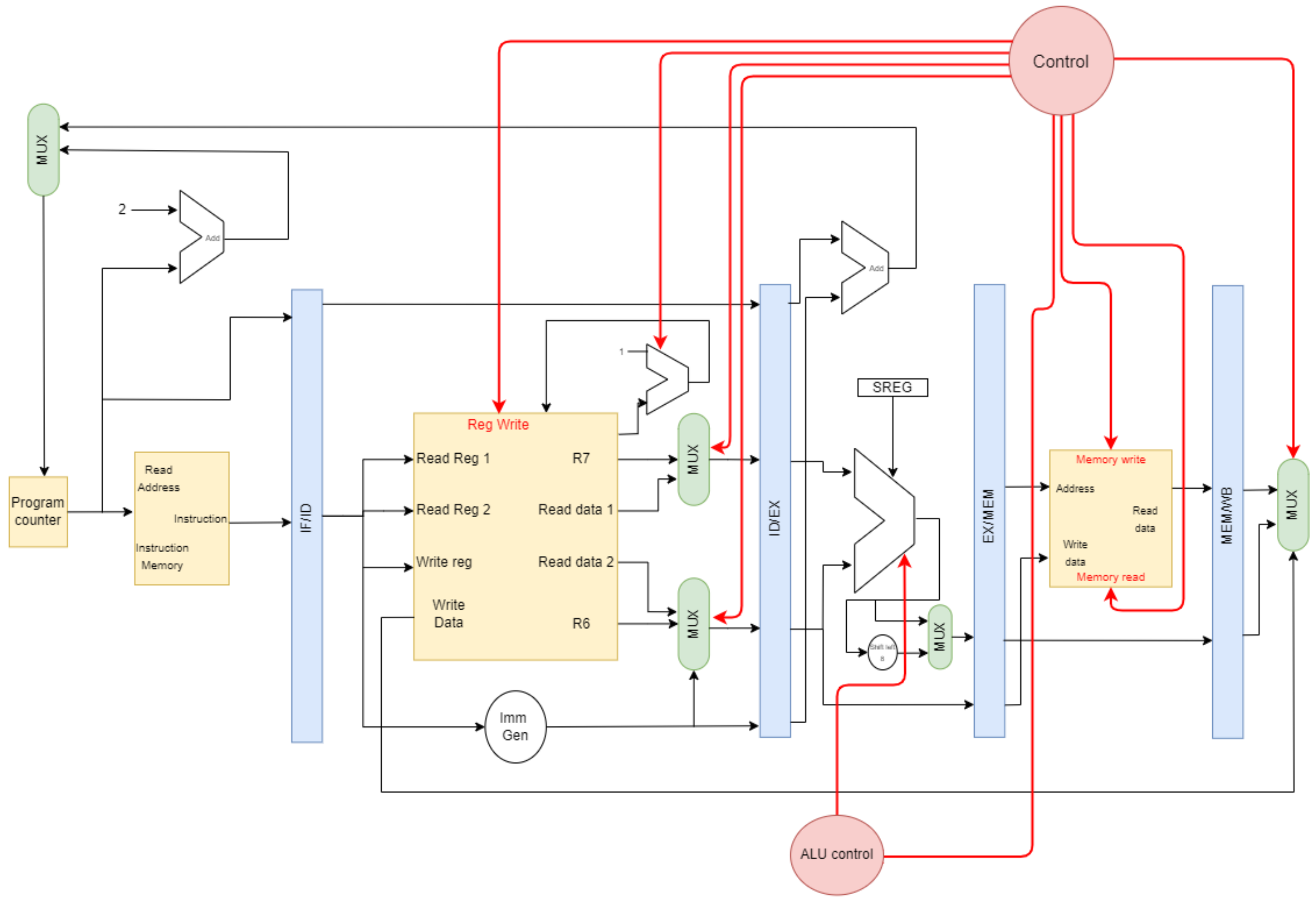
(c) Datapath


Program counter
MUX
2
Add
Read Address
Instruction
Instruction Memory
IF/ID
Read Reg 1
Read Reg 2
Write reg
Write Data
R7
Read data 1
Read data 2
R6
Imm Gen
1
MUX
MUX
ID/EX
Add
SREG
Shift left 8
MUX
EX/MEM
Address
Write data
Read data
MEM/WB
MUX

(d)

(i) The design approach that we used was very similar to the RISC-V architecture. All the information has been clearly given in each instruction (OPCODE, source and destination register addresses has been clearly illustrated). Therefore, we do not need any additional resource. Also, considering the complexity and throughput requirements, a hardwired design may be more suitable. Hardwired designs often have lower latency and can achieve high throughput for specific applications.

(ii) Controller Design



1. Instruction Fetch:
   - Control Signal: PC_MUX (Program Counter Multiplexer)
   - Justification: The PC_MUX determines the source for the Program Counter. It should have two inputs - one to accept the next sequential instruction and another to take the branch target address. This decision depends on branch instructions and is regulated by the controller.

2. Instruction Decode:
    - Control Signals: OPCODE, SRC1_MUX, SRC2_MUX, RD_MUX, ALU_OP, ALU_SRC, MEM_OP, WB_MUX
    - Justification: The controller decodes the opcode to determine the kind of instruction. Depending on the opcode, it configures the multiplexers to choose source registers (SRC1 and SRC2), the destination register (RD), sets the ALU operation (ALU_OP), defines the memory operation (MEM_OP), and configures the write-back multiplexer (WB_MUX).
3. ALU Control:
    - Control Signal: ALU_CTRL - Justification: The ALU_CTRL signal selects the precise ALU operation based on the opcode of the instruction. It's needed for arithmetic and logic instructions.
4. Memory Control:
    - Control Signals: MEM_READ, MEM_WRITE
    - Justification: These signals regulate whether a memory read or write operation should be executed. For load and store instructions, MEM_READ or MEM_WRITE will be set correspondingly.
5. Stack Pointer Update:
    - Control Signal: SP_UPDATE - Justification: In PUSH and POP instructions, the SP_UPDATE signal is utilized to increase or decrease the Stack Pointer (R7) as part of the operation.
6. Conditional Branch Evaluation:
    - Control Signals: COND_EQUAL, COND_LESS, COND_GREATER
    - Justification: For conditional branch instructions, the controller sets these signals based on the outcomes of the prior ALU operation, which is used to assess if the branch should be taken.
7. Register File Write Enable:
    - Control Signal: REG_WRITE_EN
    - Justification: This signal permits writing to the registers. It should be regulated depending on the instruction type. For example, it should be enabled for most instructions but disabled for branch instructions when no write-back happens.

8. SREG Update:

- Control Signal: SREG_UPDATE
- Justification: This signal is used to update the Status Register (SREG) with the outcomes of conditional branch assessments. It will be set when conditional branch instructions are executed.

Controller Justifications:

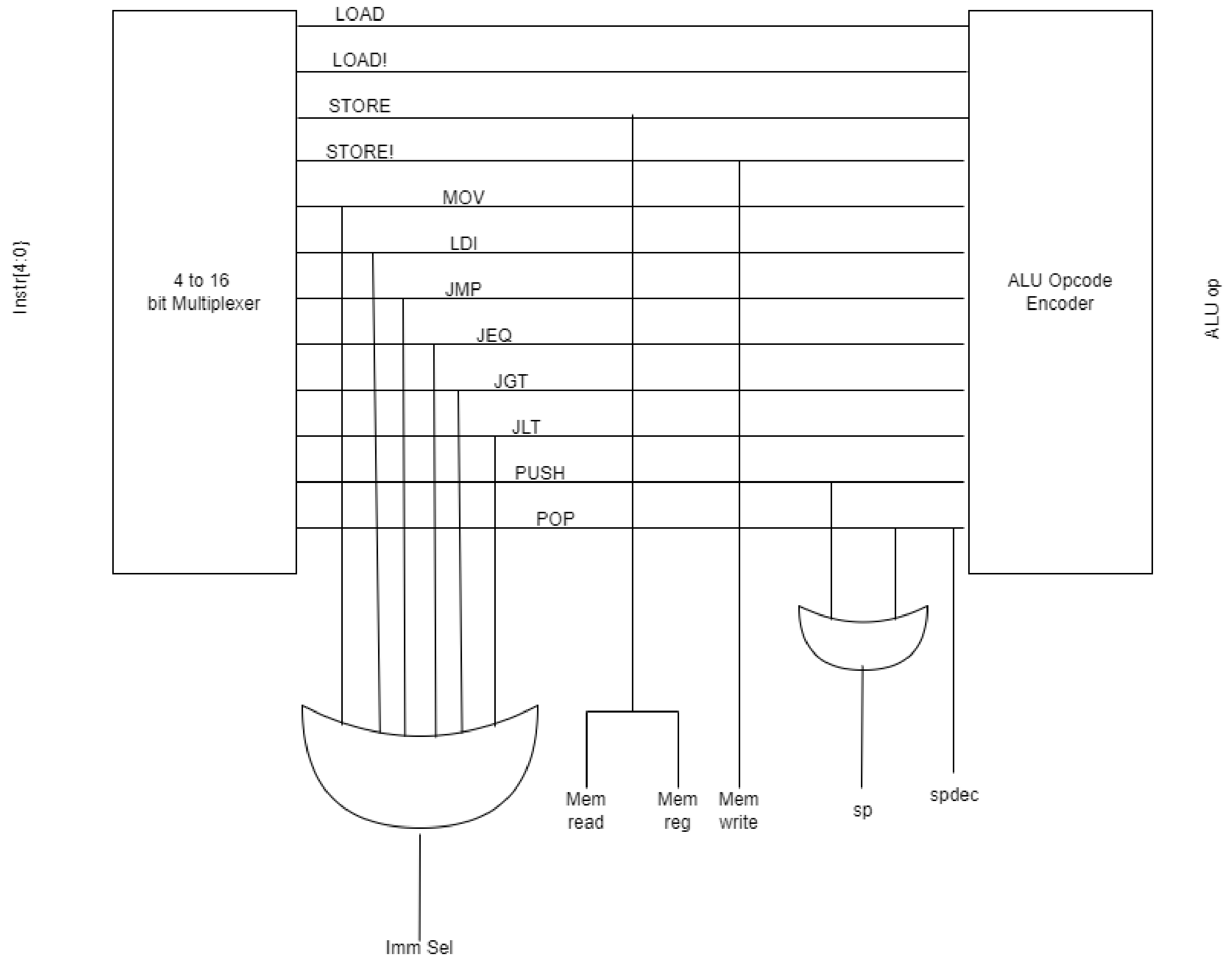
The controller design is based on the notion of a hardwired control unit. This technique is chosen because it delivers reduced latency and better throughput, which is necessary for the high-speed industrial operation as per the specifications.

- Hardwired control simplifies the architecture and minimizes the number of clock cycles required for instruction execution.

- It removes the requirement for a microprogram memory, making the CPU more efficient.
- The controller can be built as combinational logic, which is quicker than microprogrammed control.

In summary, the hardwired controller is well-suited to match the throughput needs of the bespoke processor without introducing extra complexity. It directly controls the data path components based on the opcode and other relevant circumstances.

The controller should be developed using a finite-state machine that responds to the instruction types and changes the control signals accordingly in each clock cycle.


Instr[4:0]
4 to 16 bit Multiplexer
LOAD
LOAD!
STORE
STORE!
MOV
LDI
JMP
JEQ
JGT
JLT
PUSH
POP
ALU Opcode Encoder
ALU op
Imm Sel
Mem read
Mem reg
Mem write
sp
spdec