CMPN301

Computer Architecture Project

Team 19

Spring 2021

Team Members:

Ahmed Ayman Saad 1180485
Ali Hitham Ibrahim 1180048
Eslam Ashraf Abdelaziz 1170152
Zeyad Sameh Sobhy 1180474

Instruction Format

Operation	OPCODE	X
	5 bits	11 bits
NOP	00000	00000000000
SETC	00010	00000000000
CLRC	00011	00000000000
RET	00100	00000000000

Operation	OPCODE	REG1	REG2	X
	5 bits	3 bits	3 bits	5 bits
MOV	01110	Rsrc	Rdst	00000
ADD	01000	Rsrc	Rdst	00000
SUB	01001	Rsrc	Rdst	00000
AND	01010	Rsrc	Rdst	00000
OR	01011	Rsrc	Rdst	00000

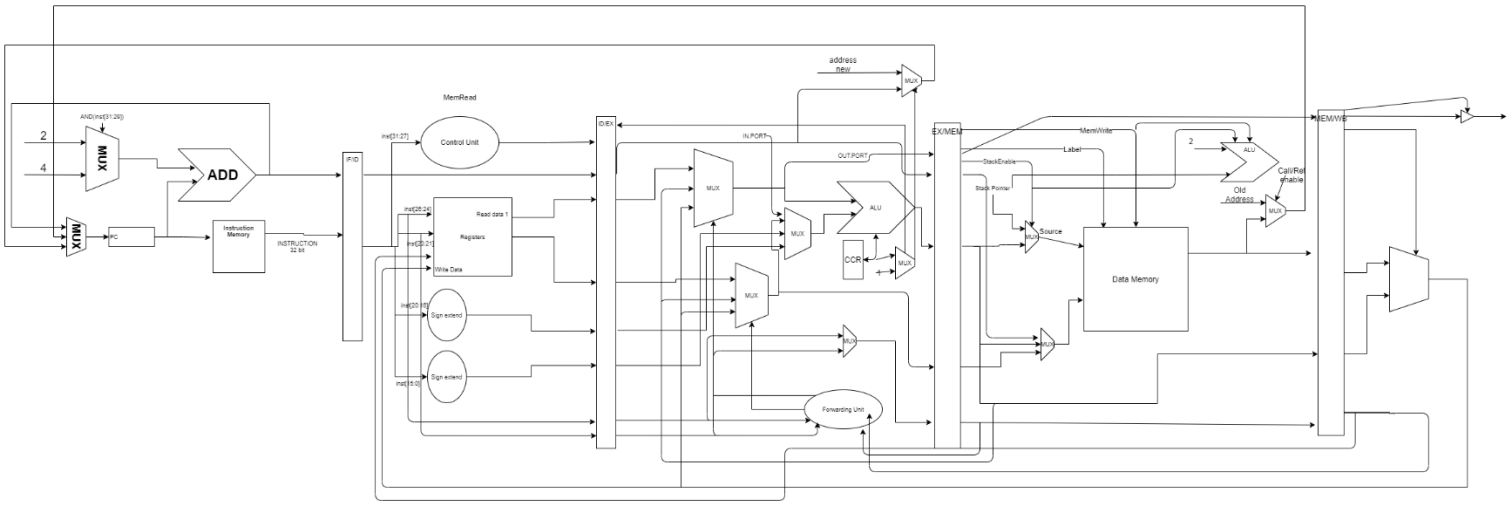
Operation	OPCODE	REG1	X	IMM
	5 bits	3 bits	3 bits	5 bits
SHL	01100	Rdst	000	IMM
SHR	01101	Rdst	000	IMM

Operation	OPCODE	REG1	X
	5 bits	3 bits	8 bits
CALL	10000	Rdst	00000000
NOT	10001	Rdst	00000000
INC	10010	Rdst	00000000
DEC	10011	Rdst	00000000
OUT	10100	Rdst	00000000
IN	10101	Rdst	00000000
PUSH	10110	Rdst	00000000
POP	10111	Rdst	00000000
JZ	11000	Rdst	00000000
JN	11001	Rdst	00000000
JC	11010	Rdst	00000000
JMP	11011	Rdst	00000000

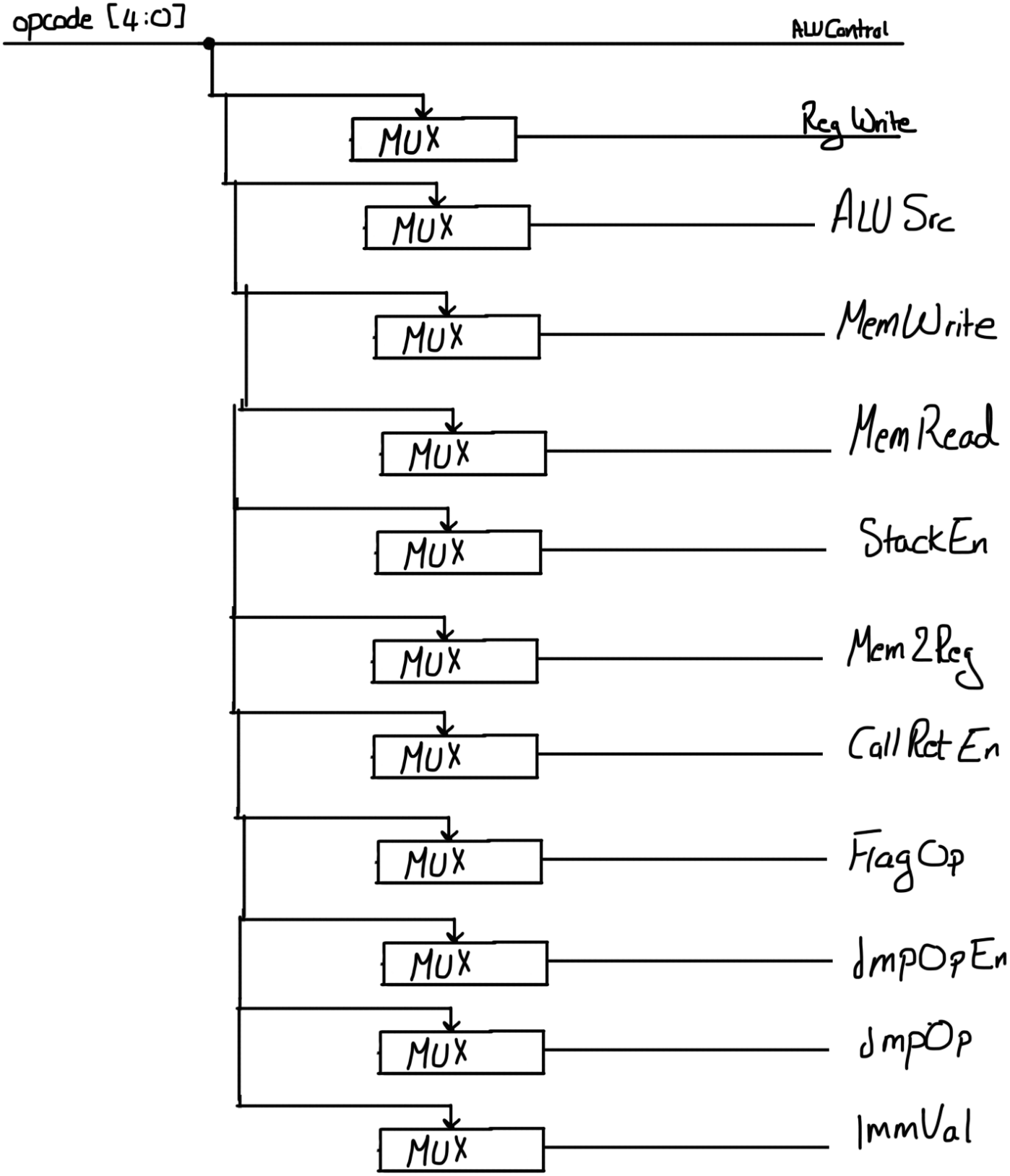
Operation	OPCODE	REG1	REG2	X	OFF
	5 bits	3 bits	3 bits	5 bits	16 bits
LDD	11100	Rsrc	Rdst	00000	OFF
STD	11101	Rsrc2	Rsrc1	00000	OFF

Operation	OPCODE	REG1	X	IMM
	5 bits	3 bits	8 bits	16 bits
IADD	11110	Rdst	00000000	IMM
LDM	11111	Rsrc	00000000	IMM

Schematic


address new
MUX
MemRead
AND(inst[31:29])
2
4
MUX
ADD
Control Unit
IF/ID
ID/EX
IN.PORT
OUT.PORT
EX/MEM
MEM/WB
MemWrite
Label
StackEnable
Stack Pointer
ALU
Call/Ret enable
Old Address
MUX
Source
Data Memory
MUX
PC
Instruction Memory
INSTRUCTION 32 bit
inst[31:27]
inst[26:24]
inst[23:21]
Read data 1
Registers
Write Data
inst[20:16]
Sign extend
inst[15:0]
Sign extend
MUX
CCR
MUX
MUX
MUX
MUX
Forwarding Unit

Control Unit detailed design



Instruction	OPCODE	RegWrite	ALUSrc	ALUControl	RegDst	MemWrite	MemRead	StackEn	DataSrc	Mem2Reg	CallRetEn	FlagOp	JmpOpEn	JmpOp	ImmVal
NOP	00000	0		00000	X	0	0	0		0	0	0	0		
SETC	00010	0		00010	X	0	0	0		0	0	1	0		
CLRC	00011	0		00011	X	0	0	0		0	0	1	0		
RET	00100	0		00100	X	0	1	1		0	1	0	0		
MOV	01110	1		01110	Rdst	0	0	0		0	0	0	0		
ADD	01000	1	01	01000	Rdst	0	0	0		0	0	0	0		
SUB	01001	1	01	01001	Rdst	0	0	0		0	0	0	0		
AND	01010	1	01	01010	Rdst	0	0	0		0	0	0	0		
OR	01011	1	01	01011	Rdst	0	0	0		0	0	0	0		
SHL	01100	1	10	01100	Rdst	0	0	0		0	0	0	0		b10
SHR	01101	1	10	01101	Rdst	0	0	0		0	0	0	0		b10
CALL	10000	0		10000	X	1	0	1		0	1	0	0		
NOT	10001	1		10001	Rdst	0	0	0		0	0	0	0		
INC	10010	1		10010	Rdst	0	0	0		0	0	0	0		
DEC	10011	1		10011	Rdst	0	0	0		0	0	0	0		
OUT	10100	0		10100	X	0	0	0		0	0	0	0		
IN	10101	1	00	10101	Rdst	0	0	0		0	0	0	0		
PUSH	10110	0		10110	X	1	0	1		0	0	0	0		
POP	10111	1		10111	Rdst	0	1	1		1	0	0	0		
JZ	11000	0		11000	X	0	0	0		0	0	0	1	b00	
JN	11001	0		11001	X	0	0	0		0	0	0	1	b01	
JC	11010	0		11010	X	0	0	0		0	0	0	1	b10	
JMP	11011	0		11011	X	0	0	0		0	0	0	1	b11	
LDD	11100	1	11	11100	Rdst	0	1	0		1	0	0	0		b01
STD	11101	0	11	11101	X	1	0	0		0	0	0	0		b01

IADD	1111 0	1	11	11110	Rdst	0	0	0		0	0	0	0		b01
LDM	1111 1	1	11	11111	Rdst	0	0	0		0	0	0	0		b01

Pipeline registers details

Fetch:

- **Instruction Memory**
- **Data Memory**
- **Program Counter**
- **PC Index Selector**
- **Reset Signal**
- **IN Port**

-Buffer:

- **Instruction**

  Output: Separated into different bits

Decode:

- **Registers Block**

  Input: source and destination registers
  Output: data from source and destination registers

- **Control Unit**

  Input: op code
  Output: (ALUSrc)Enable for immediate value (bits 2 and 3 11)

  Output: Memory Read Enable
  Memory Write Enable (last 4 bits bits opcode)

  Output: Number of write register

  Output: Memory to Register (in case of load)

  Output: Register Write Enable (write back to register in WB stage)

  Output: Enable Stack

- **Condition Code Register**
- **New instruction address calculation**
- **Hazard Detection Unit**

  Input: Memory read enable from execute buffer, register source , register destination, op code
  Output: Signal to delete control unit signal buffer, sigal to pc counter to stop index, blocks decode part of cycle

-Buffer:

- Register Destination Data
- Chosen Data
- CCR
- New instruction Address
- Execute Buffer:
  ALU Control
- Memory Buffer:
  Memory Read Enable
  Memory Write Enable
  Stack Enable
- Write Back Buffer:
  Memory to Register
  Number of write register
  Register Write Enable

Execute:

- **ALU Block**
- **ALU Control (selector of operation)**
- **Forwarding Unit**

  Inputs: Decode buffer source register
  Decode buffer destination register
  Execute buffer source register
  Execute buffer destination register
  Memory buffer source register
  Memory buffer destination register
  Output: Enable Register source
  Enable Register destination

- **MUX source**

  Input: data source, alu data result from execute buffer, alu data result from memory buffer

- **MUX destination**

  Input: data dest, alu data result from execute buffer, alu data result from memory buffer

-Buffer:

- ALU Result
- Register Destination data
- CCR (flag register)
- New instruction Address
- Memory Buffer:
  Memory Read Enable
  Memory Write Enable
  Stack Enable
- Write Back Buffer:
  Memory to Register
  Number of write register
  Register Write Enable

Memory:

- **Data Memory**

-Buffer:

- ALU Result
- Register Destination data
- CCR (flag register)
- Write Back Buffer:
  Memory to Register
  Number of write register
  Register Write Enable

Write Back:

Selector (to write back to Register File)

Pipeline Hazards

There are several hazards that may arise due to pipelining the processor. Structural hazards arise from the need to use the same component at the same time. However, in this design, this is solved by having a separate memory each for data and instruction. Also, no component can be accessed by two different instructions at the same time. Data hazards result from the dependencies of some instruction on the result of a previous instruction. Data hazards can be solved by either forwarding which is to forward the result from the ALU as soon it is available or by stalling the pipeline when we have no other option. In such cases, each stage of the pipeline design is stalled preventing any execution of another fetched instruction. This is the case usually with load instruction as the value is not available until after the fourth stage. Stalling mean we perform no operation in the stalled cycle, no new instruction is loaded, and the previous instructions are preserved in the pipeline registers.

Hazard Resolution Forwarding

To solve data hazards, a comparison is done in the pipeline registers to determine the presence or absence of a hazard. We have two situations where we can do the forwarding: the first is to forward the result from the Ex/Mem Register to the ALU, and the second is to forward the result from Mem/Wb register to the ALU. Forwarding is implemented because we can't wait for the register to be written to in the last stage

Data Hazard Detection

For the detection of data hazard, as earlier stated, there are two paths for forwarding. To detect a hazard in the Ex stage, we compare the EX/MEM destination register against both ID/EX source registers that has been read. If either comparison condition is true, we forward the result from the prior ALU to the next ALU as an operand. If the condition is false, then the operands are passed from the register files.

Similarly, for the second path, we check MEM/WB destination register against both ID/EX source registers. If the condition is true, the result value stored in the Mem/Wb register will be forwarded.

Important considerations:

1- We need to ensure that the instruction does a write back to a register. We achieve this by checking the register write signal (RegWrite), as this signal is only enabled when it's '1'.

2. We want to avoid forwarding the non- zero value of $zero, if it was set as destination register. MIPS Architecture defines $zero as a register that is hardwired to value '0'. It can be used as a target register whose value is to be discarded. This can be achieved by checking the destination register value for zero.

3. In the case of MEM/WB forwarding, we have an additional hurdle that might occur if we use the same destination for several instructions. In this case the forwarding will be

inaccurate because we want to forward the result of the most recent instruction, not the one before.

Stalling Instructions

The other method is to stall the pipeline because we can't produce the required operand on time for the ALU to operate on it. In a case of Load instruction followed by an instruction that uses the loaded value as one of its operand, the hazard detection unit identifies that the instruction requires a stall and it stalls the pipeline by inserting a 'bubble'. This is done by zeroing the ID/EX register including the control signals. Therefore, no memory or registers are written if the control values are zeros.

6.1. Hazard Detection Unit

Now, we can identify whether a stall is required at the decode stage if the instruction is a load instruction and it causes a conflict with next instruction. In this case, we cannot use only the load opcode. The instruction is load if the memory to register signal is set (MemtoReg = '1') and if the destination register for that instruction is equal to one of the operands of the next instruction, there is a load hazard condition.