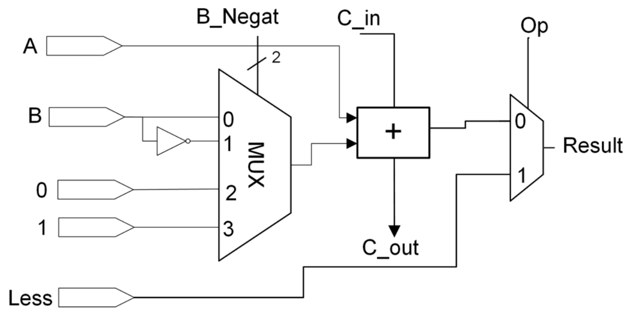
2017_1_A1

Design a 32-bit ALU for the following operations: ADD, SUB, INC, DEC, SLT. Design the 1-bit ALU by using only one adder and the minimum number of components and control signals. Extend the 1-bit ALU to obtain the 32-bit ALU. Show the schematic with control signals and a table with the control signal values for the required operations.

ADD → a + b	→ C_IN = 0
SUB → a – b → a + NOT(b) + 1	→ C_IN = 1
INC → a – 1 → a + 0 + 1	→ C_IN = 1
DEC → a – 1 → a + NOT(1) + 1	→ C_IN = **0**

NOT(0) → extended on 32 bits only → 0xFFFFFFFF

0x78 –	0b0111 1000 -	0b0111 1000 +	→	0b0111 1000 +	
0x01	0b0000 0001	0b1111 1110 +		0b1111 1111	
		0b0000 0001		0b0111 0111	→ 0x77



1-bit ALU (for 32-bit ALU see lecture 5, cascade 32 times, MSB result from adder of ALU_31 connected to Less input of ALU_0, rest of Less signals connected to 0)

Operation	B_Negat	C_in	Op
ADD	00	0	0
SUB	01	1	0
INC	10	1	0
DEC	11	0	0
SLT	01	1	1

2017_2_A1: ADD, SUB, INC, DEC, SLT, AND, OR, NAND, NOR.
NAND → De Morgan → NOT(A) OR NOT(B)
NOR → De Morgan → NOT(A) AND NOT(B)

2017_1_B1

Design a 16-bit ALU for ADD, SUB, INC and DEC, for 16-bit operands or 2 x 8-bit operands (MMX style). Draw the schematic with control signals using two 8-bit adders and the necessary auxiliary circuits. Show a table with the values of the control signals for the required operations.

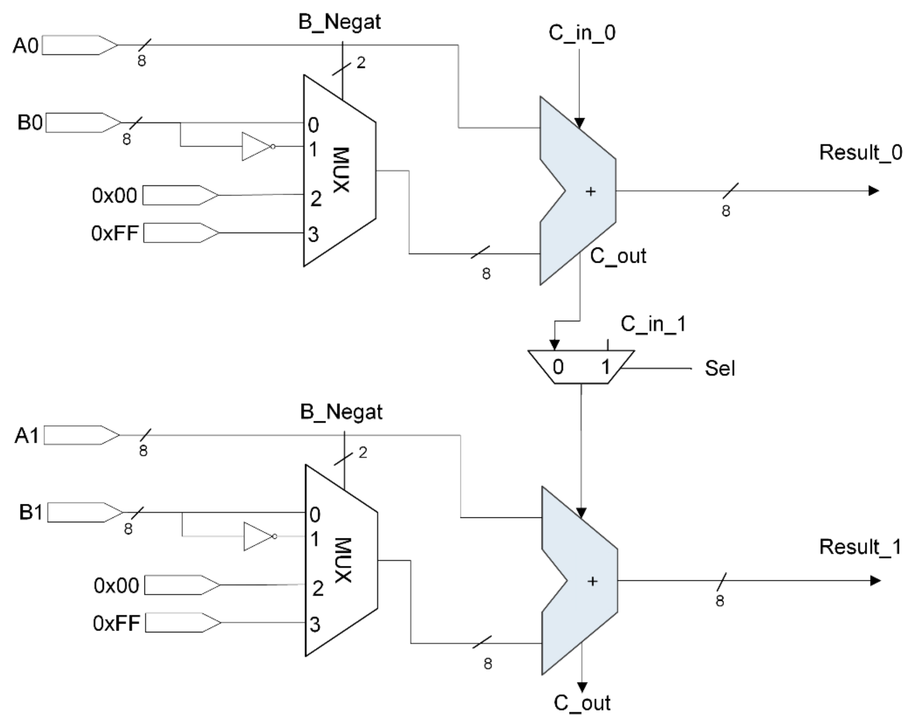
ADD → a + b → C_IN = 0

SUB → a – b → a + NOT(b) + 1 → C_IN = 1

INC → a – 1 → a + 0x00 + 1 → C_IN = 1

DEC → a – 1 → a + 0xFF + 0 → C_IN = **0**

Operation	B_Negat	C_in_0	C_in _1	Sel
ADD – 2x8-bits	00	0	0	1
SUB – 2x8-bits	01	1	1	1
INC – 2x8-bits	10	1	1	1
DEC – 2x8-bits	11	0	0	1
ADD – 16 bits	00	0	X	0
SUB – 16 bits	01	1	X	0
INC – 16 bits	10	1	X	0
DEC – 16 bits	11	0	X	0



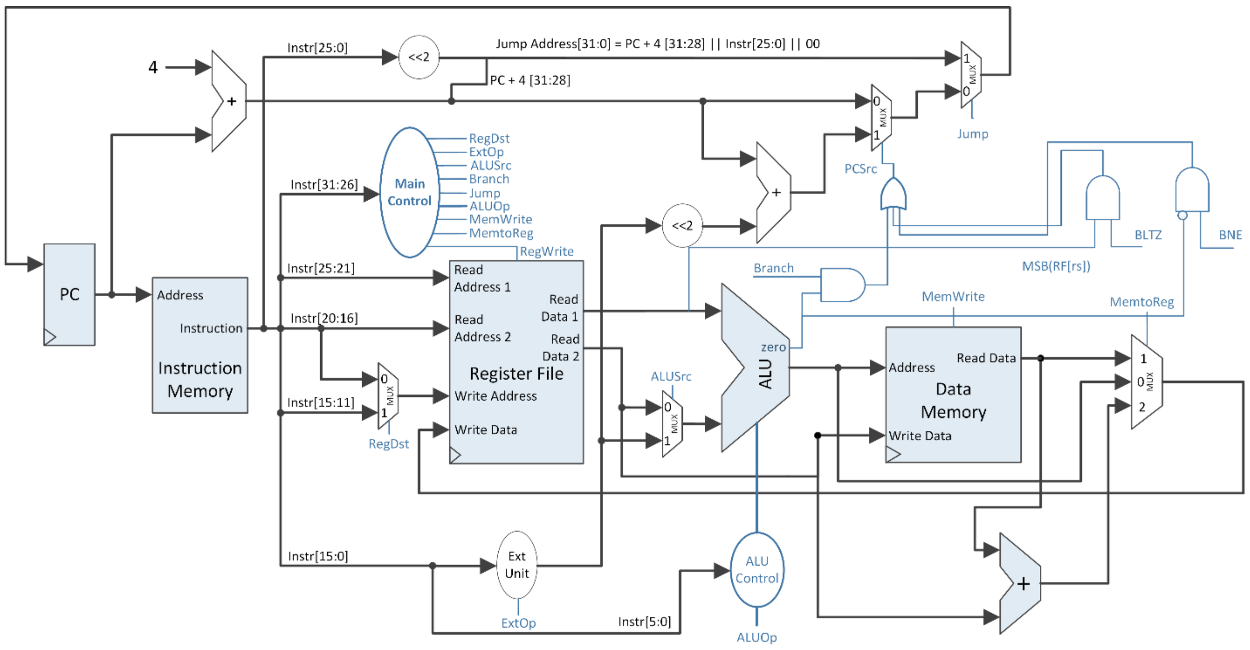
Problems: Add instructions in Single-Cycle MIPS.

BLTZ (branch on less than zero) → I-type instruction
bltz rs, imm
RTL abstract:
If (RF[rs] < 0) then PC ← PC + 4 + Sign_Ext(Imm << 2);
Else PC ← PC + 4
Added a control signal specific for BLTZ instruction. In order to differentiate from other branch instructions we add an OR gate before the PCSrc control signal. The OR gate inputs are the normal branch decision and the new bltz decision.

BNE (branch on not equal) → I-type instruction
bne rs, rt, imm
RTL abstract:
If (RF[rs] != RF[rt]) then PC ← PC + 4 + Sign_Ext(Imm << 2);
Else PC ← PC + 4

ADDM (add memory: adds the content of a register to a memory location and writes the result in the Register File) → I-type instruction
RTL abstract:
addm rs, rt, imm
RF[rt] ← M[RF[rs] + Sign_Ext(Imm)] + RF[rt]; PC ← PC + 4;

- Added a 32-bit adder for the computation
- Extended the MemToReg MUX to 3-inputs.



Single-cycle data-path extended for BLTZ, BNE, ADDM

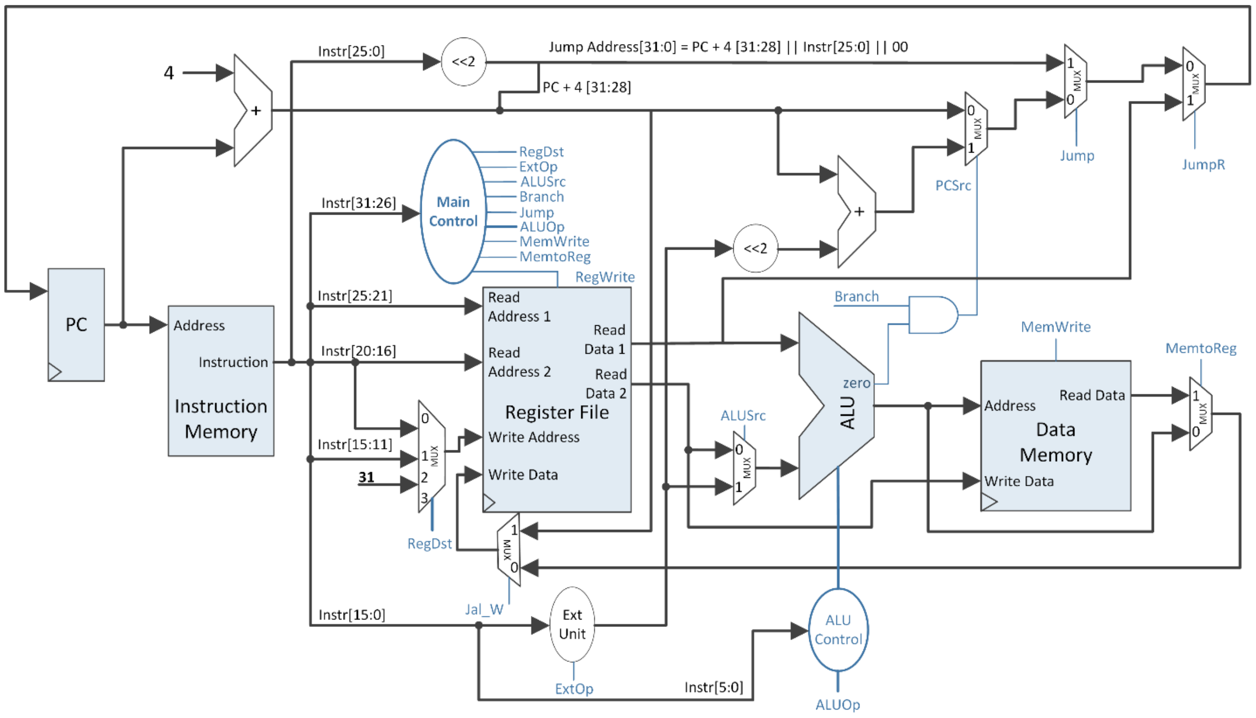
Instruction	Reg Dst	Reg Write	Ext Op	ALU Src	ALU Op	Mem Write	Memto Reg	Branch	BNE	BLTZ	JUMP
bltz	X	0	1	X	XX	0	XX	0	0	1	0
bne	X	0	1	0	01 (-)	0	XX	0	1	0	0
addm	0	1	1	1	00 (+)	0	10	0	0	0	0

Usually we treat X (don't care) as 0.

Problems: Add instructions in Single-Cycle MIPS:
JAL (jump and link) instruction and JR (jump register).

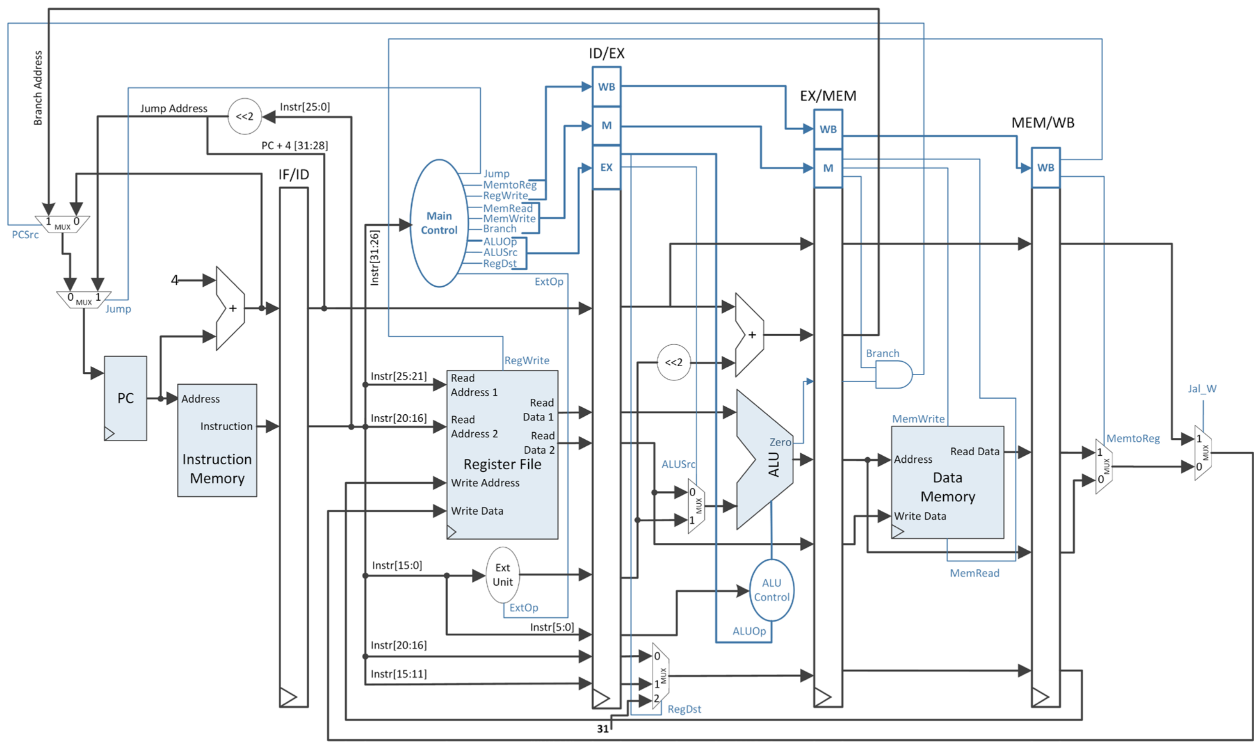
JAL RTL abstract
Single cycle: PC ← PC[31:28] || target_address || 00; RF[31] ← PC + 4;
Pipeline: PC ← PC[31:28] || target_address || 00; RF[31] ← PC + 8; (1 instruction will enter in the pipeline until the jump target address is known)

JR RTL abstract
Single cycle: PC ← RF[rs];



Single cycle data-path extended for JAL and JR instructions

Instruction	Reg Dst	Reg Write	Ext Op	ALU Src	ALU Op	Mem Write	Memto Reg	Branch	JUMP	Jal_W	JumpR
Jal	10	1	X	X	XX	0	X	0	1	1	0
jr	XX	0	X	X	XX	0	X	0	0	X	1



Pipeline data-path extended with JAL instruction

Cache memory problems

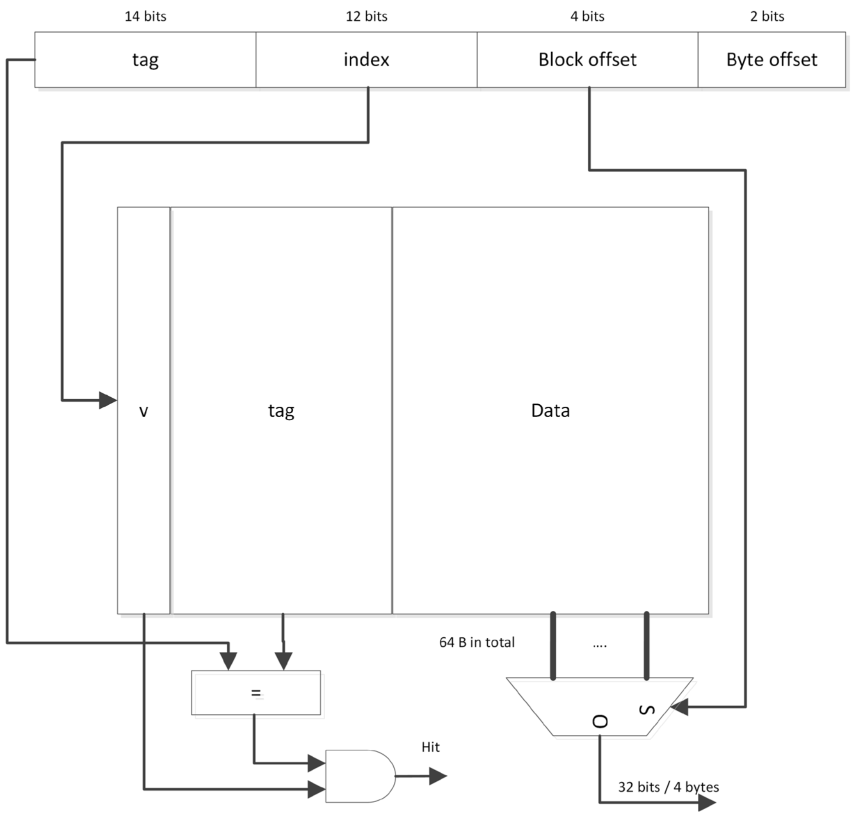
A 32-bit CPU generates 32-bit addresses for a byte addressable memory. Design a **256KB** cache memory for this CPU. The block size is 64 bytes. Show the block diagram, and the address decoding for a direct mapped cache memory and a 4-way set associative cache memory.

Direct Mapped

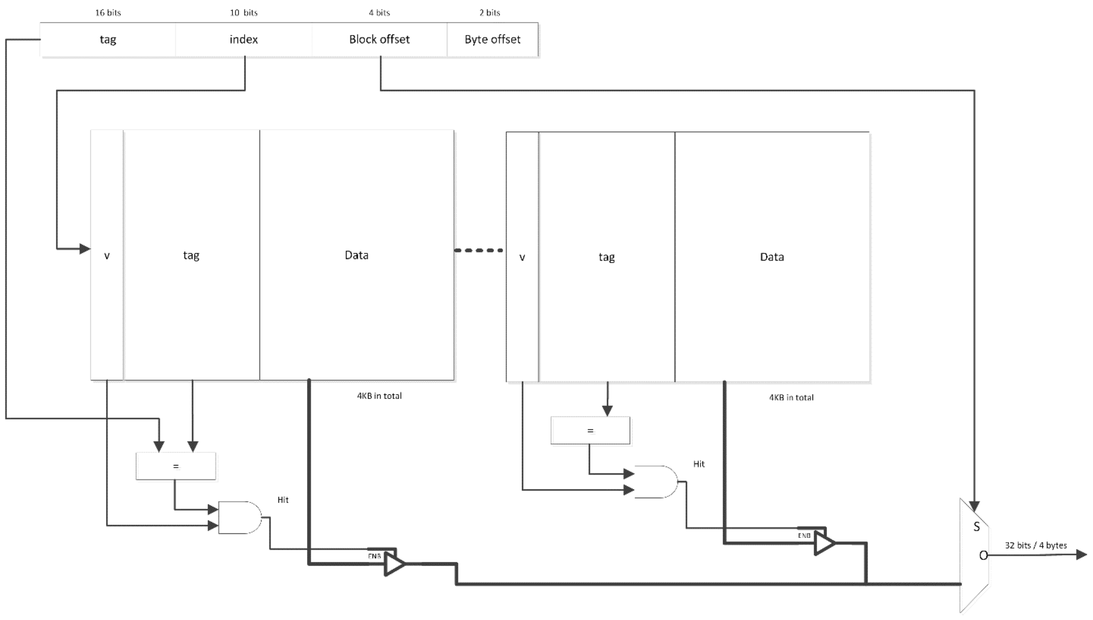
256 KB = 2^8 x 2^{10} B = 2^{18} B	→ 18 bits address
Block size = 64 bytes = 2^6 bytes	→ 6 bits address
Data 32-bits = 4B = 2^2 bytes	→ 2 bits – byte offset
No of blocks = 256KB / 64B = 2^{18}/ 2^6= 2^{12} blocks	→ 12 bits – block index
Block offset = 2^6 / 2^2 = 2^4	→ 4 bits – block offset
Tag: 32 – 12 – 4 – 2	→ 14 bits – tag

4-way set associative cache

256 KB = 2^8 x 2^{10} B = 2^{18} B	→ 18 bits address
Block size = 64 bytes = 2^6 bytes	→ 6 bits address
Data 32-bits = 4B = 2^2 bytes	→ 2 bits – byte offset
No. of sets = 4 → 256 KB / 4 = 64 KB	
No of blocks = 64 KB / 64B = 2^{16}/ 2^6= 2^{10} blocks	→ 10 bits – block index
Block offset = 2^6 / 2^2 = 2^4	→ 4 bits – block offset
Tag: 32 – 10 – 4 – 2	→ 16 bits – tag



Direct mapped cache

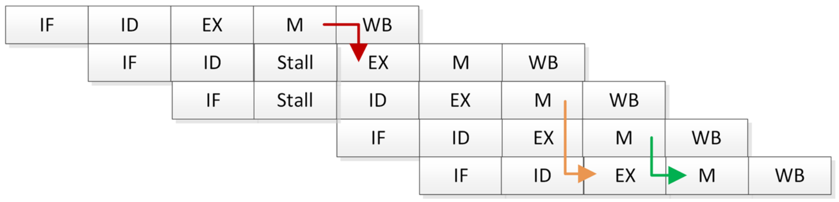


4-way set associative cache

Pipeline diagrams:
Draw the pipeline diagram with and without forwarding for the following code sequence: lw $6, 20($7); add $8, $6, $3; or $1, $6, $6; add $8, $3, $2; sw $8, 20($1). Identify and explain the hazards in the code sequence and how they are resolved. Specify how many clock cycles does it take to execute the code sequence in the two configurations.

1. lw $6, 20($7);	RF[$6] ← M[RF[$7] + 20]
2. add $8, $6, $3;	RF[$8] ← RF[$6] + RF[$3]
3. or $1, $6, $6;	RF[$1] ← RF[$6] + RF[$6]
4. add $8, $3, $2;	RF[$8] ← RF[$3] + RF[$2]
5. sw, 20($1)	M[RF[$1] + 20] ← RF[$8]

RAW: Instr 2 with Instr 1 → $6
resolved by forwarding from Mem stage of Instr1 to EX stage of Instr2 + 1 stall
RAW: Instr 3 with Instr 1 → $6
No longer a hazard – WB for $6 in first half of clk cycle 5, read in ID in second half of clk cycle 5
RAW: Instr 5 with Instr 3 → $1
Resolved by forwarding from M stage of Instr 3 to EX stage of Instr 5
RAW: Instr 5 with Instr 2, 4 →$8
Resolved by forwarding from M stage of Instr 4 to M stage of Instr 5
10 clock cycles to execute



IF	ID	EX	M	WB					
	IF	ID	Stall	EX	M	WB			
		IF	Stall	ID	EX	M	WB		
				IF	ID	EX	M	WB	
					IF	ID	EX	M	WB

Pipeline diagram with forwarding