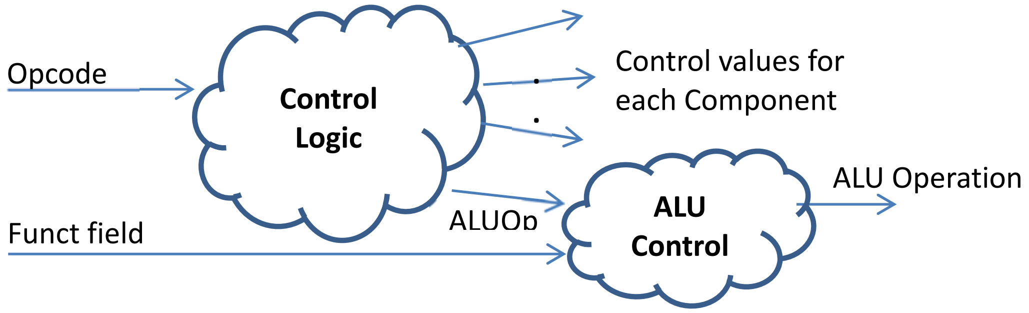
ALU Control

- Depending on the instruction the ALU is used in different capacities:
  - Load/Store: add
  - Branch on Equal: subtract
  - R-type: depends on funct field
- Because the ALU control is based on not only the opcode, but the function field (R-type), the control logic is split into 2 steps.

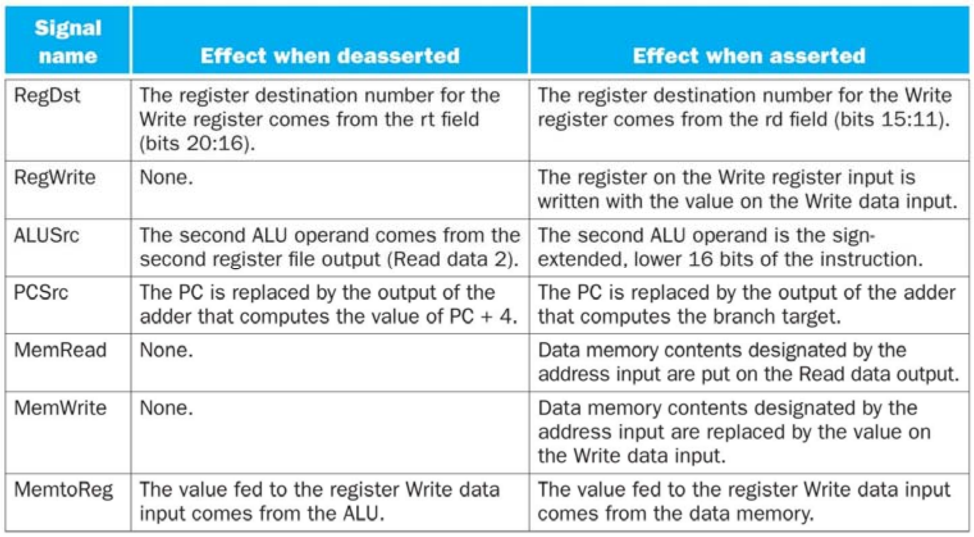


- Based on the opcode a 2-bit control value, ALUOp, can be created. This value then combined with the function field produces the proper ALU Operation control.

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

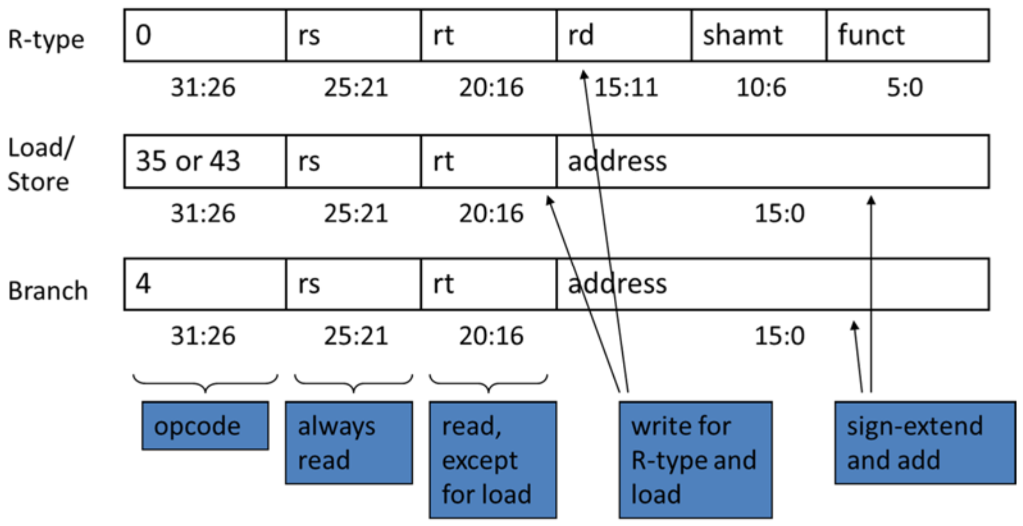
Control Unit

- The remaining control logic is generate by the Control Unit
- All of the logic is basic gates
- We define the remaining signals as follows (asserted = 1, deasserted = 0)
- Note that PCSrc is the combination of the Branch control and the Zero output of the ALU combined together



Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

- Another way to consider the control values is to look at the instruction formats:

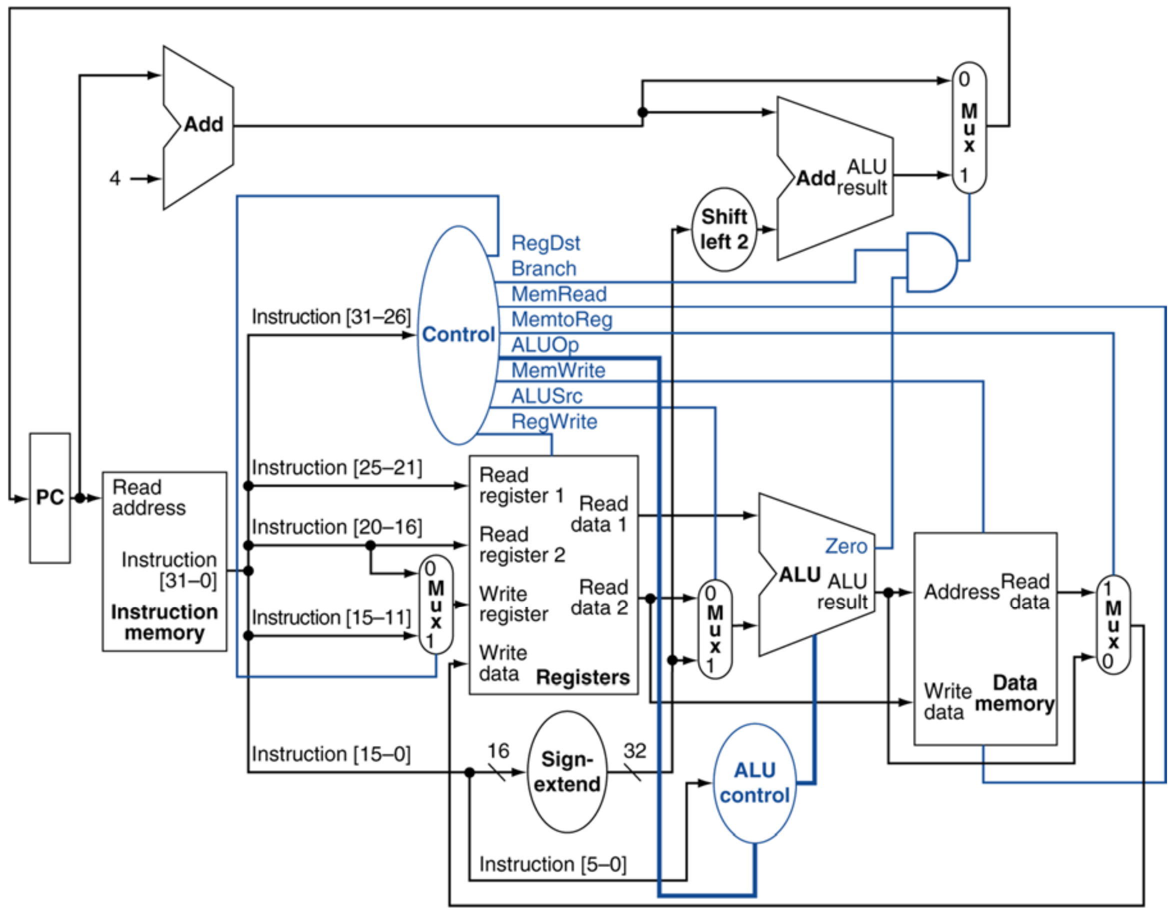


- The control table for the current instruction sets as follows:

Instr	**Opcode**	**RegDst**	**RegWrite**	**ALUSrc**	**Mem Read**	**Mem Write**	**MemtoReg**	**Branch**	**ALUOp**
lw	100011	0	1	1	1	0	1	0	00
sw	101011	X	0	1	0	1	X	0	00
R-type	000000	1	1	0	0	0	0	0	10
beq	000100	X	0	0	0	0	x	1	01

Singe Cycle Datapath

- Below the current datapath for the instructions above.
- The blue lines represent all the control signals of the datapath.



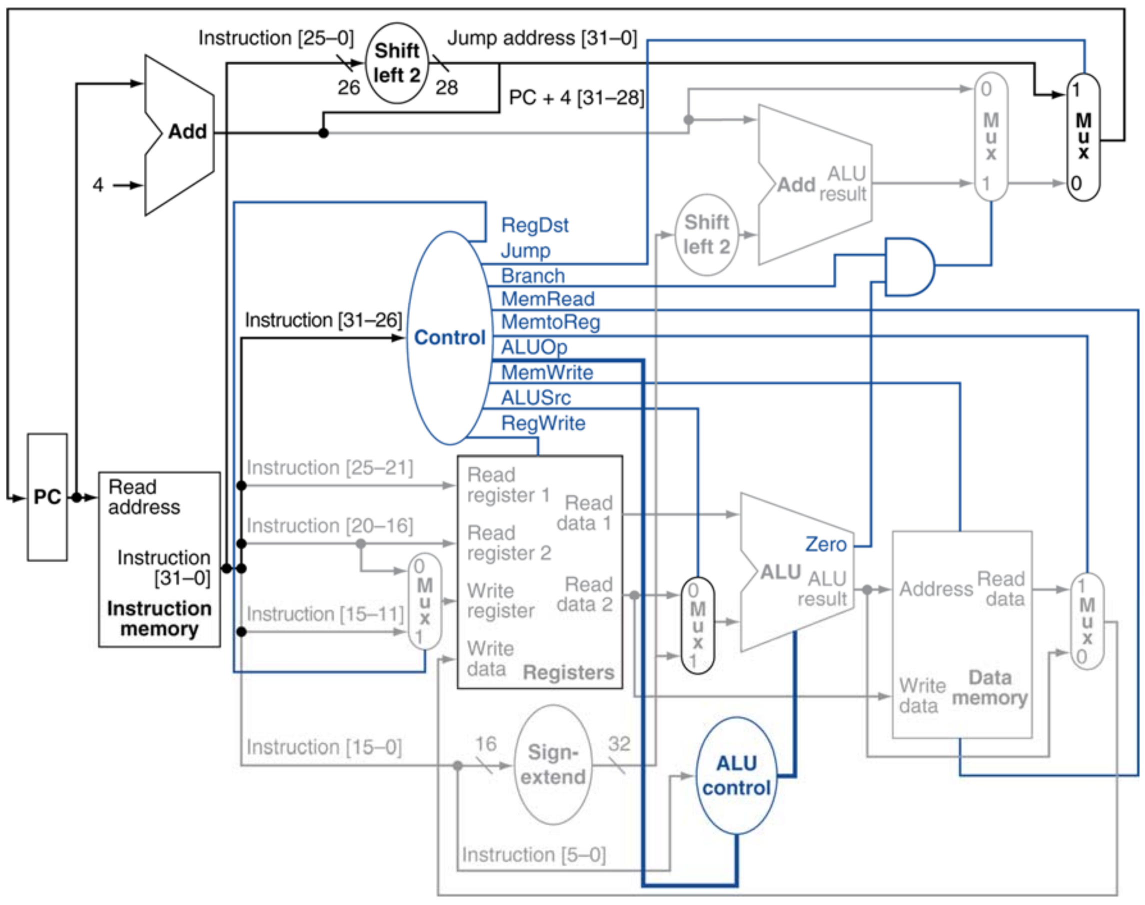
Jump Instructions – separate "special" group

- Jump instructions perform an unconditional change to the PC based on the value in the instruction
- The destination address for a jump instruction is the concatenation of:
    - upper 4 bits of the current PC + 4
    - 26-bit address field in instruction
    - 00 as the 2 low-order bits (word address, not byte. Therefore multiply by 4.)

Field	000010	address
Bit positions	31:26	25:0

- Extra control signal is required based on opcode in order to support jump

Instr	Opcode	RegDst	RegWrite	ALUSrc	Mem Read	Mem Write	Memto Reg	Branch	ALUOp	Jump
lw	100011	0	1	1	1	0	1	0	00	0
sw	101011	X	0	1	0	1	X	0	00	0
R-type	000000	1	1	0	0	0	0	0	10	0
beq	000100	X	0	0	0	0	X	1	01	0
Jump	000010	X	0	x	0	0	X	X	XX	1



Side Note: Review on the difference between Branch and Jump instructions

Branch is (PC+4) + (sign extended immediate value *4) where the immediate value is the Number of instructions from the beq (current instruction) - 1. The minus 1 is because at fetch we automatically do the PC +4, so when executing the branch you've already gone forward one instruction. So to branch 2 instructions forward the immediate value in the instruction should be 1. To move backward 3 instructions the immediate value should be -4.

The jump is different in the sense that we are not modifying the PC with respect to how many instructions fwd/bkwd we want to move. For jump the full 32bit address can not be stored in the instruction itself (need 6 bits for opcode). Therefore we can only specify 26 bits of the jump address in the instruction. We also know that all instructions are on word boundaries (multiple of 4) and therefore the 2 least significant bits are always 00. We can use this to our advantage and not represent these 2 0's in the instruction (waste of space, also allows for larger range of addresses to be represented/jumped to). So, if we specify the 26 bits in the instruction and then shift left by 2 (mult by 4) we have 28bits of the new address. We are still missing 4 bits, these must come from the current PC, where else could they come from... So the jump address is the top 4 bits of current PC+4, PC[31...28] appended to the 28bits (jump address in instruction shifted left by 2).

This means when you use a jump instruction you can only jump to an address in the same section of memory. The upper 4 bits of the PC divide the memory into 16 different sections, and jump only allows you to move to any address within your current section. To jump to another section you need to use jump to register (which assumes the full address to jump to is stored in the register).