ICS53 - MIPS Instruction Set Architecture & Single-Cycle Datapath (continued)

Branch on Equat

Ex: `beq $t0, $s0, label     # if Reg[$t0] == Reg[$s0], then  PC = PC + 4 + label`

[**Branch on Equal Instruction Walk-through**]

I-type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

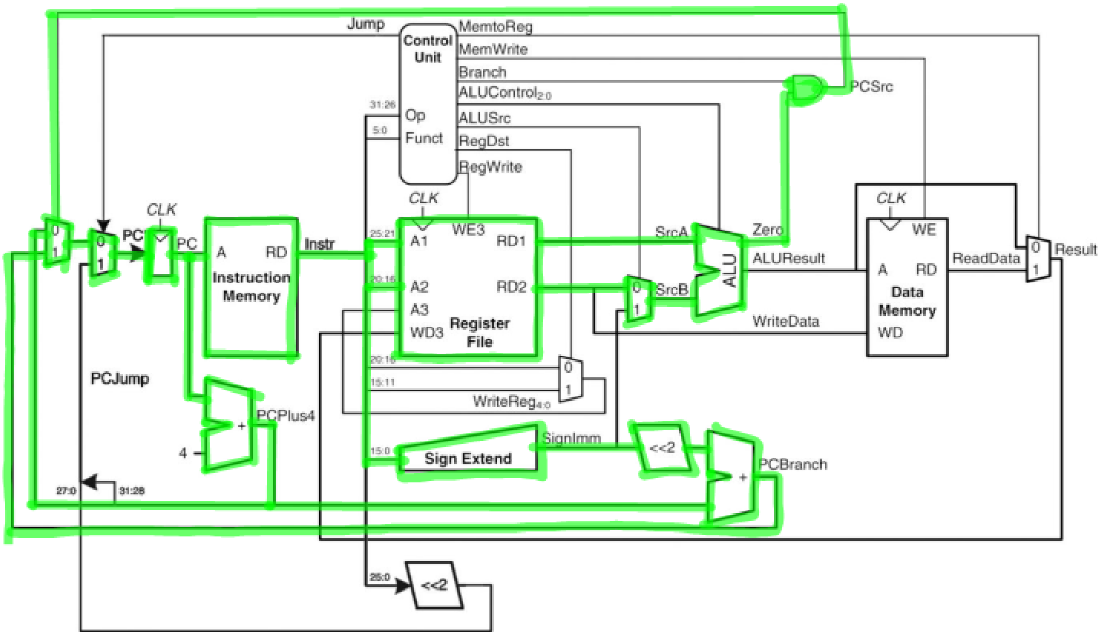
- These instructions take two operands from registers and perform an ALU comparison operation on them, if the operation is "true" then the PC is replaced with the location of the next instruction to execute.
- The label value in the immediate field from the instruction, is not the 32-bit address of the label. Instead the field represents +/- the number of instructions from the PC+4 address.
  - Most branch statements are local jumps for loops and conditional checks. These jumps are not to far of places in memory. Therefore, +/- 2^{15} instructions is more than enough range.
  - Note, how it is the number of instructions, not the number of bytes. Since all label addresses are multiples of 4, we can increase the range by eliminating the need to store the LSB 00 in the immediate field. Instead we shift the immediate field to the left by 2 to multiple the value by 4.
  - Note, the calculation of the relative position of the branch label is performed based on the PC+4 value, not the current instruction. The reason for this will become clear in the next module. For now, just accept this fact.

Branch

Fetch – Fetch Instruction
Decode – Register Read (rs: `$s0`, rt: `$t0`), Sign Extend Immediate Value
Execute – Compare operands by Performing subtract ALU operation (Reg[$s0] – Reg[$t0])
Execute – If "true" (Zero ==1), set PC value to target address

- In the execute stage, we subtract the register values and check to see if the result is 0 using the Zero output of the ALU. Remember, the Zero output has a value of 1 if the result of the ALU is 0, 0 otherwise.
- While the operands are being compared, the target address (the location of the next instruction to execute if branch is "true") is calculated by:
  - Sign-extend the immediate value (positive or negative 2's complement value)
  - Shift left 2 places (multiply by 4 to get word address)
  - Add value to PC + 4 value

- Note that for the branch operation, we use the PCBranch adder to calculate the branch address.
  - This is because the entire datapath must execute in a single clock cycle. This means that each component of the datapath can only perform 1 operation. **Hardware cannot be reused in the same instruction during the clock cycle.**



Jump

Ex: `j label        # PC = PC+4[31:28],label[25:26] – comma represent concatenation`

Jump Instruction Walk-through

J-type

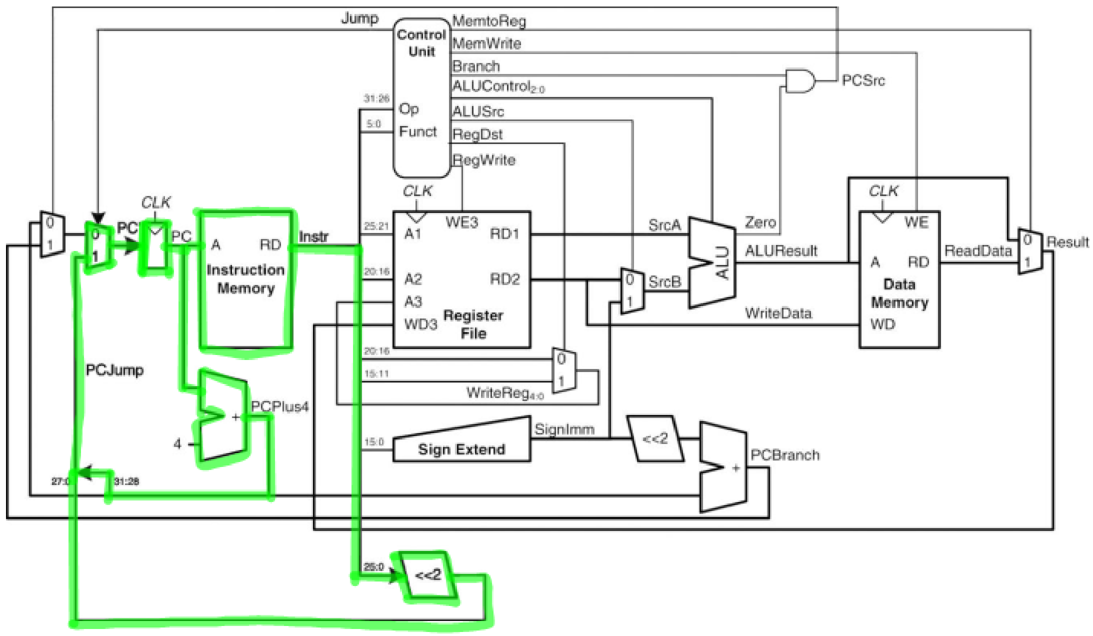
op	addr
6 bits	26 bits

- Jump instructions perform an unconditional change to the PC based on the value in the instruction
- Jump is different than a branch in the sense that we ARE NOT modifying the PC with respect to how many instructions forward and backward. A portion of the absolute address to jump to is provided in the addr field.

- The full 32-bit address cannot be stored in the instruction itself (need 6 bits for opcode, it won't fit!). Therefore, we can specify at most 26-bits of the jump address.
  - o All instructions must be stored on word boundaries of memory (multiple of 4). Therefore, the 2 least significant bits of the address are always going to be 00.
  - o We can use this to our advantage and not store these zeros 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 (multiply by 4), we now have 28-bits of the jump address.
  - o We are still missing 4 bits. Where can we get these? Since when we jump, we are most likely jump to a location close to the current instruction, then we can take these bits from the current PC (same region of memory)
    - ▪ 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. Jump instructions only allows you to move to any address within your current section. To jump to another section, you need to use `jr` (jump to register)
      - `jr` jumps to the full 32-bit address stored in the `rs` register
  - o The jump address is the top 4 bits of current PC+4, specifically PC[31...28] appended to the 28-bits (jump address in instruction shifted left by 2).

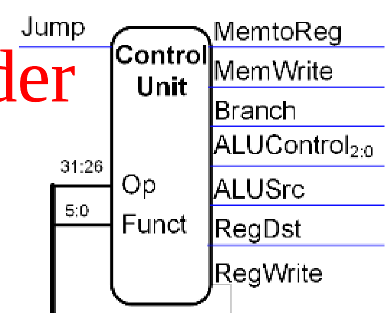
- To summarizes, the destination address for a jump instruction is the concatenation of:
  - o upper 4 bits of the current PC + 4

- o 26-bit address field in instruction
- o 00 as the 2 low-order bits (word address, not byte. Therefore, multiply by 4.)



Control Unit

[Single-Cycle Control Unit](#)



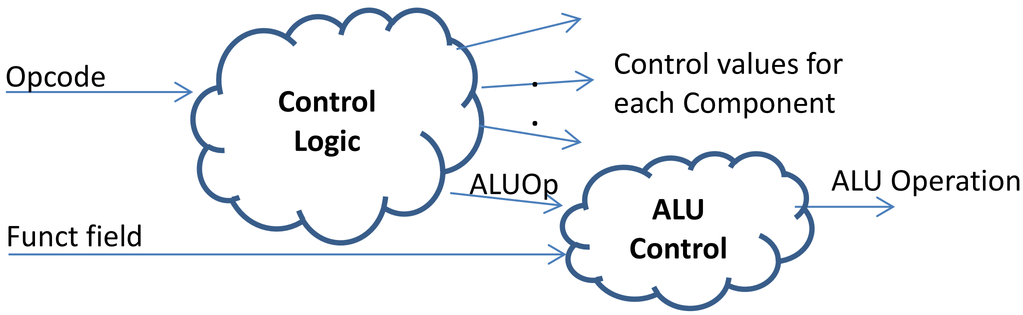
- The names of the control signals are named to represent flags, eg. Branch = 1 means branch is true/taken, Branch = 0 means branch is false/not taken.
  - o A signal is asserted = 1 or deasserted = 0

- The complete Control Table the 5 basic operations implemented in the datapath is shown below:

Instr	Opcode	RegDst	RegWrite	ALUSrc	MemRead	MemWrite	MemtoReg	Branch	ALUOp	Jump
lw	100011	0	1	1	1	0	1	0	00 (add)	0
sw	101011	X	0	1	0	1	X	0	00 (add)	0
R-type	000000	1	1	0	0	0	0	0	10 (func)	0
beq	000100	X	0	0	0	0	X	1	01 (sub)	0
jump	000010	X	0	X	0	0	X	X	XX	1

- Note the ALUOp column of the table. This value is a 2-bit control determined within the Control Unit and is used to create the 3-bit ALUControl output for the ALU.
- Because the ALU control is based on not only the opcode, but the function field (R-type), the control logic unit is split into 2 steps.
  - o This reduces the number of inputs for the ALUOperation Boolean expression from 12-bits (6 for opcode and 6 for func field) to 8-bits (2 for intermediate ALUOp and 6 for func field)

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



ALUOp	Meaning
00	add
01	subtract
10	look at **funct**
11	not used

- When the opcode is lw or sw, ALUOp is set to 00 to denote addition is performed. For beq, ALUOp is set to 01 to denote subtraction is performed. The function field in these cases is ignored (aka Don't care), as the Immediate instruction format do not contain a function field.
- In the case of R-type instructions, the function field is the unique identifier for the operation (remember that the opcode is set to all zeros in the case of R-type instructions).
    - The ALUOp is set to 10, to denote the function field should be considered when determining the ALUControl.

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	010
sw	00	store word	XXXXXX	add	010
beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
		subtract	100010	subtract	110
		AND	100100	AND	000
		OR	100101	OR	001
		set-on-less-than	101010	set-on-less-than	111

- !! In all of the discussions/practice problems/notes we use the 2-bit ALUOp to denote the operation performed by the ALU unit instead of the 3-bit ALUControl (less to remember).