Merging datapaths: (`add`,`lw`, `sw`)

The datapaths that we saw last lecture considered each instruction in isolation. I drew only those elements that were needed for each instruction. Of course, the various wires and inputs are hardware and thus they are present in every instruction. In this lecture we merge the various datapaths. We do so using multiplexors to select from different possible inputs for circuit elements, for example, to select which values are to be written into registers or Memory on each clock cycle. We also need to say more about control signals, for example, the selector signals for the multiplexors.

The figure on the following page shows how the `add`, `lw`, `sw` datapaths from last lecture are merged into a single circuit. I have not draw a single "control unit", but instead I have just shown the various control lines (in green) where they are needed. Let's discuss these control signals one by one. I'll begin by discussing the controls of the three multiplexors, shown in blue.

- RegDst: Recall the R and I format instructions have the form:

bits	26-31	21-25	16-20	11-15	6-10	0-5
R-format	op	rs	rt	rd	shamt	funct
numbits	6	5	5	5	5	6

I-format	op	rs	rt	immediate
numbits	6	5	5	16

  For either R or I format instructions, at least one register is *read*. For this reason, one of the register slots in the instruction format is always used for a read. This is the *rs* slot, namely bits [21-25]. This 5 bit line is always fed into the ReadReg1 input and specifies which register we read from.

  It is not true, however, that an instruction always *writes* to one of the 32 registers. For example, `sw` do not write to a register. The *rt* slot, namely bits [16-20], holds a second register number which `add` and `sw` read from, and which `lw` writes to.

  If an R-format instruction uses three registers (`add`), then two of them are read from and one of them is written to. The *rd* register (R format) is the one that is written to.

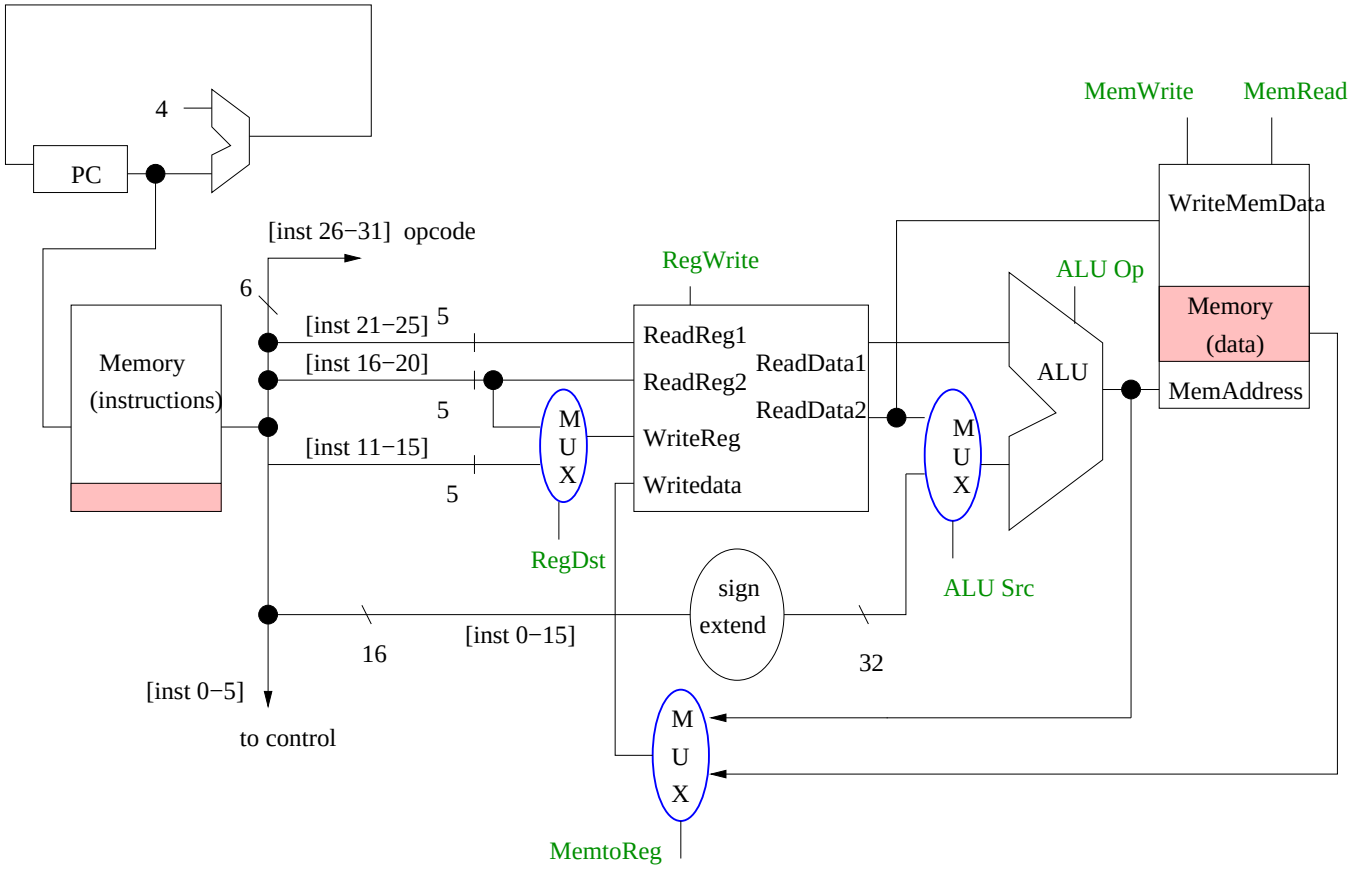
  To summarize, *rs* is always specifies a read register ("s" standing for source), *rd* always specifies a write register ("d" standing for destination) and *rt* is a register that is either read from or written to, depending on the instruction. (I am not sure that "t" stands for anything. Maybe "t" is the letter after "s" and rt is the slot after rs.)

  If you look at the merged datapath, you see that the three 5-bit register number lines are routed to the register array and that a multiplexor is used to decide which line goes to the WriteReg input. The selector signal for this multiplexor is called RegDst (register destination); the value is 1 means rd is selected and otherwise the value is 0.

- ALUSrc: One input of the ALU always comes from a register. So one of the ReadData registers (namely ReadData1) is hardwired to one of the ALU inputs. The other ALU input comes from either a register (in the case of an `add`) or from the signed-extended 16 bit immediate field (in the case of a `lw` or `sw`). Thus a multiplexor is needed to choose from among these

two possible inputs to the second ALU input. The selector for this multiplexor is ALUSrc. I will say more about the ALU controls later.

- MemtoReg: The third multiplexor shown in the figure selects where the register write value comes from. It might come from the ALU (R-format) or it might come from Memory `lw`.



Other controls are shown in the merged data path. RegWrite specifies whether or not we are writing to a register. MemWrite specifes whether or not we are writing to Memory. MemRead specifies whether or not we are reading from Memory. (The reason for needed two separate controls will be more clear later on, when we look at the mechanics of memory access.) Note that the RegWrite signal was called[1] "WriteEnable" in lecture 6 page 5. The MemWrite signal was shown in lecture 6 p. 6,7.

	input	output (of control, i.e. control variables)						
	opcode	RegWrite	MemWrite	MemRead	MemToReg	RegDst	ALUsrc	ALUOp
R-format	000000	1	0	0	0	0	1	10
lw	100011	1	0	1	1	1	0	00
sw	101011	0	1	0	X	X	0	00
bne	000100	0	0	0	0	X	X	00

[1] I called it that because there was another input line with a similar namely, namely WriteReg and I didn't want you to get confused.

The ALUOp control variable needs more explanation. You may have assumed that it specifies which operation the ALU should perform. For example, `add, sub, slt, and, or` all require different operations to be performed by the ALU. Recall that the operation is determined by the instructions function field, since all R-format instructions have the same opcode, so the function field will also be needed as a input to the circuit that computes the control signals. Here is how it works.

If the ALUOp control is 10, this means that the instruction is R-format and therefore the operation to be performed by the ALU is determined by the funct field. As you recall from Lecture 4, the ALU circuit has two control inputs. There is a Binvert which is used for subtraction, and that takes the twos complement of the B variable. And there is a coded operation (either AND, OR, or +). Since there are three operations to be code, we need two bits. We called this control "op" to distinguish it from "ALUOp."

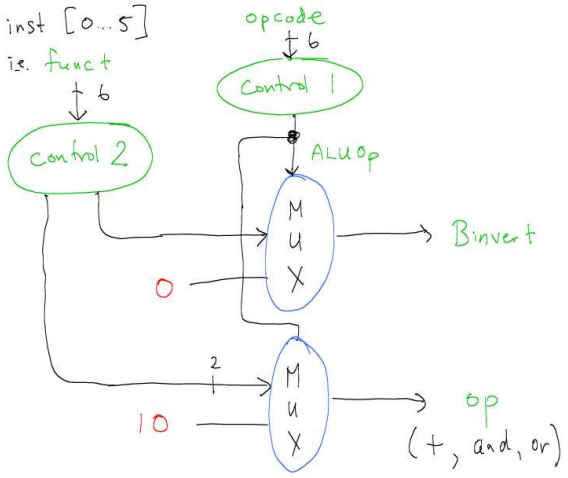
Let's suppose that the op control for the ALU uses 10 for addition. Then this is what the lw and sw instructions also need to use for their op control. The add and sub instructions also use the 10. (The difference between add and sub comes from the Binvert control, not the op control.) If the instruction is `and` then op gets code 00 and if the instruction is `or` then op gets 01.

There is one more op value that could be used: 11. This is used for slt. Set less than needs to perform a subtraction. If the result of the subtract is negative, then the result of the instruction which is written back into a register is a 1, and otherwise a 0 is written back into a register.

Here is the truth table that specifies how to take inputs (ALUOp and funct) and compute the control signals needed by the ALU.

			ALU controls	
inst	ALUOp	funct	Binvert	op
`lw`	00	xxxxxx	0	10
`sw`	00	xxxxxx	0	10
`bne`	01	xxxxxx	1	10
`add`	10	100000	0	10
`sub`	10	100010	1	10
`and`	10	100100	0	00
`or`	10	100101	0	01
`slt`	10	101010	1	11

Notice that the control units can be computed in two steps. The first step computes ALUOp. If ALUOp is 10, then we have R-format instruction and the funct fields are used to compute the the controls Binvert and op which determine what the ALU computes. Otherwise, the opcode can be used directly to determine Binvert and op. We can therefore decompose the control unit into two parts, as illustrated by the circuit below. Note that the control ALUOp is used as a selector. *This is not quite what is illustrated in the figures on pages 2 and 5, but I left those figures as they were to keep the clutter down.*

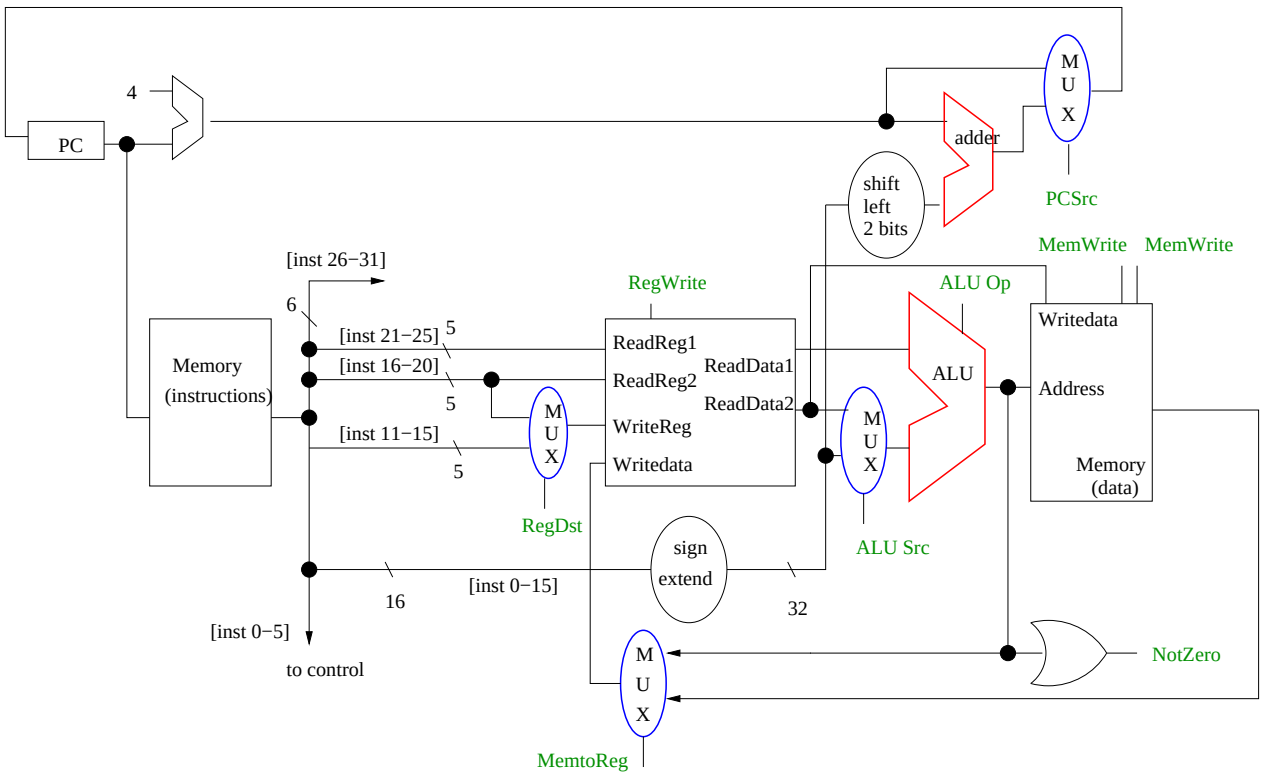


Merging the datapaths for (`bne`)

Let's consider how we can merge in the `bne` datapath that we saw last class. We need a multiplexor to select between PC+4 and PC+4+offset. Call this selector signal PCSrc. In general, the value of PCSrc depends on the instruction (`bne` vs. `add`, etc) as well as on the value of other variables such as NotZero (in the case of `bne`). There are other possibilities to update the PC, for example, if there is an unconditional jump instriction (see below). So we need to build a small control unit that determines PCSrc. This control unit would need the opcode [inst 26-31] and other inputs (for the case of conditional branches).

Notice that we use a separate adder circuit to compute the branch address for the `bne` instruction, rather than the ALU. The reason that we need an adder here is that the ALU is already being used in this instruction, namely to check whether the two arguments have the same value.

The merged datapath is shown on the next page.



Datapath for jal (jump and link)

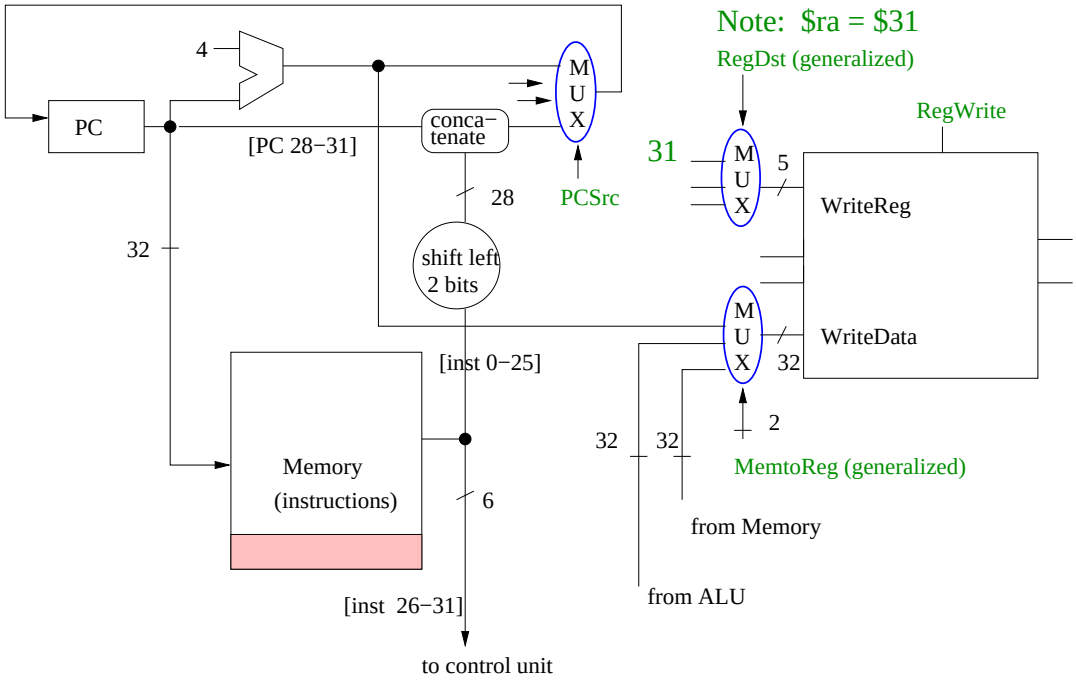
Last lecture we looked at the jump instruction `j`. Let's look at a related instruction: jump and link `jal`. This instruction also has J format:

field	op	address
numbits	6	26

Like the jump instruction `j`, the jump and link instruction jumps to an address that is determined by a concatenation of the upper 4 bits of the PC, the lower 26 bits of the instruction, and the bits 00 which go into the two lowest order address bits (because instructions are word aligned).

The difference between `jal` and `j` is that `jal` also writes PC+4 into register $ra, which is $31. I have written this 5 bit values as 31 in the figure. I have also added a line which shows how PC+4 gets written into register $31.

Notice in the figure that the two controls WriteDataSrc and MemtoReg are now "generalized" All I mean by that is that previously these controls were 1 bit because they had to choose between two possibilities, but now there are three possibilities and so they need to be two bits. (I could have changed the names of these controls to something more general, but we won't need them further so there is no need for that.) bother.)



Datapath for jr (jump register, R format)

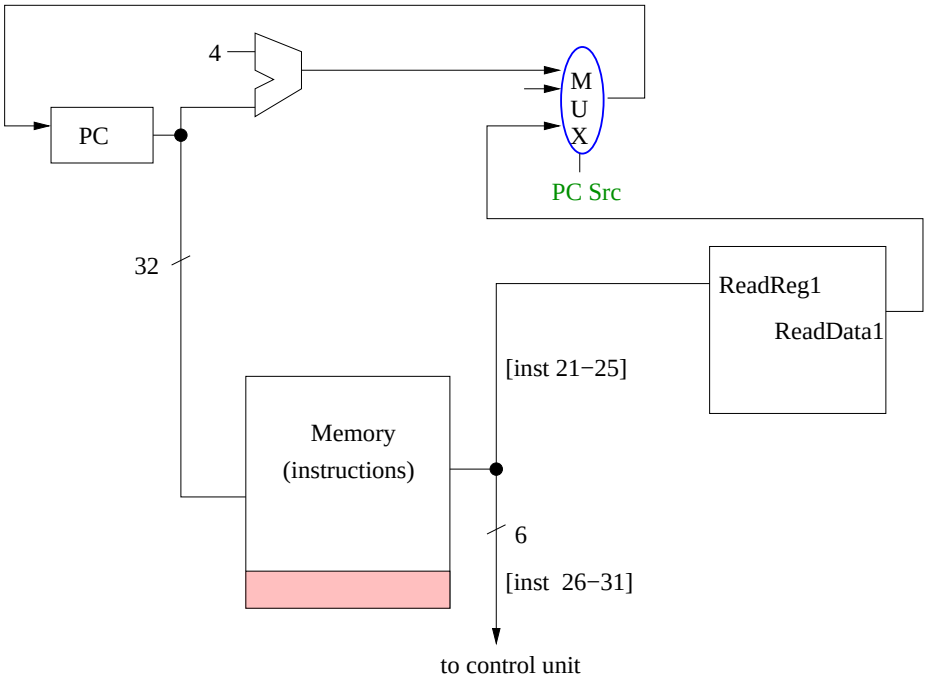
You may be surprised to discover that the `jr` instruction has R format. From our arguments at the beginning of this lecture, you can see why. The instruction `jr $r*` (typically `$ra`) is encoded by inputting the register number (e.g. `$ra` = 31) into ReadReg1. The address to jump to is read from this register. Usually this is the return address of a function. This return address is read out from the register and selected as the value to be written into the PC, at the end of the clock cycle.

Since we are reading from a register, the register number is in the *rs* field. The *rt*, *rd*, and shamt fields contain junk for this instruction. The funct field does not contain junk though. Since this is an R-format instruction, the opcode is the same as for the `add` and other instructions, and so `jr` is only distinguished from these other R-format instructions by the funct field).

Summary of Single Cycle implementation

What we have described is a *single cycle implementation*. The entire instruction (including the instruction fetch, the update of the PC, the ALU operation, the read or write from memory) is done in one clock cycle. This is possible because the datapaths for these instructions never involve involve more than one write into register or memory. Careful here: you can write into the PC and into a register within one instruction, but you are using two different paths within that instruction. The combinational circuits within these two paths run in parallel.

As mentioned earlier, in a single cycle implementation, the interval between clock pulses must be long enough to make sure that all the signals stablize by the end of the clock cycle (e.g. the 32 bit ALU requires all the Carryin signals to ripple through...) If a word is read from Memory, the value must be available on the output wires from the Memory. Since the clock pulses occur at a regular rate, we must design for the worst case.



What is the worst case? For the instructions we discussed above, the worst case is the `lw` instruction. We need to read from registers, to do an addition, read from Memory, and then to write the data item into a register. For the `add` instruction, you don't need to use the Memory at all, so this is much faster.

Designing for the worst case is a big problem, in terms of efficiency. And in fact, real MIPS computers do not use a single cycle implementation. Rather they use multiple cycles. Next lecture, I will introduce some ideas of how a multiple cycle implementation could work.

The idea of using multiple clock cycles should not be entirely new to you. We have already discussed how integer multiplication and division require a sequences of shifts and either additions or subtractions, and that floating point operations require multiple shifts, compares (of exponents), adds and subtracts, etc. Such instructions require their own control circuits and "micro-sequences". Next class I will go over the basic principles of how this can be done.