You are familiar with how MIPS programs step from one instruction to the next, and how branches can occur conditionally or unconditionally. We next examine the machine level representation of how MIPS goes from one instruction to the next. We will examine how each MIPS instruction is "decoded" so that data is passed to the appropriate places (e.g. from register to memory, from memory to a register, from a register to another register, etc). By the end of this lecture and the next, you will have a basic understanding of how the combinational and sequential circuits that you learned at the beginning of this course are related to the MIPS instructions and programs that we have covered in more recent lectures.

Data paths for MIPS instructions

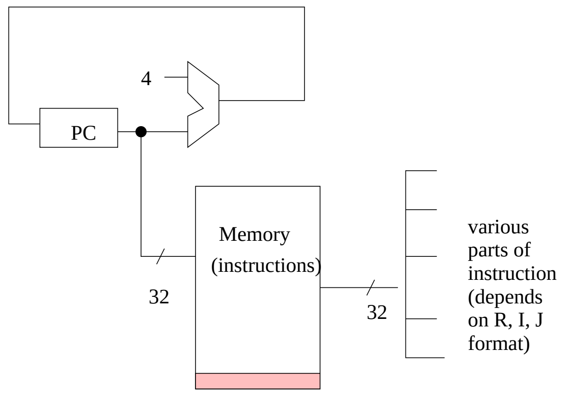
In this lecture and the next, we will assume that one instruction is executed in each clock cycle. This is known as the *single cycle* model. In the lecture following the study break, we will see the limitations of the single cycle model, and we will discuss how MIPS gets around it, namely by "pipelining" the instructions.

Instruction fetch

Both data and instructions are in Memory. For an instruction to be executed, it first must be read out of Memory. MIPS has a special *program counter* register (PC) that holds the address of the current instruction being executed. As a MIPS programmer, you are not responsible for "fetching" the next instruction from memory. This done automatically. Let's have a look at how this is done.

Consider the simple case of a non-branching instruction such as `add` or `lw` or `sw`. The address of the instruction that follows this non-branching instruction is, by default, the address of the current instruction plus 4 (that is, plus 4 bytes or one word). Because we are assuming that each instruction takes one clock cycle, at the end of clock cycle, PC is updated to PC+4.

The current instruction (`add` or `lw` or `sw`, ...) is fetched by using PC to select a word from the text part of Memory, namely the word containing the current instruction. Later in the course, we will see how this is done. For now, it is enough for you to understand that an address is sent to Memory and the instruction at that address is read out of Memory. To summarize, the contents of the PC (a 32 bit address) is sent to Memory and the instruction (also 32 bits) starting at that address is read from Memory.



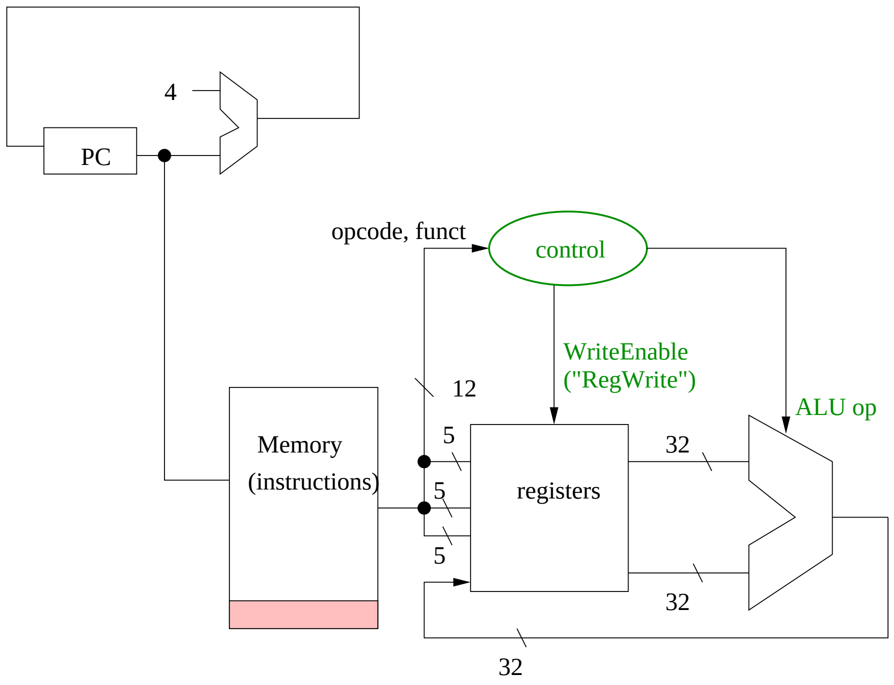
Let's next look at several examples of instructions and consider the "datapaths" and how these are controlled.

Data path for `add`

What happens when the 32 bit `add` instruction is read out of memory? Recall that an `add` instruction has R format:

field	op	rs	rt	rd	shamt	funct
numbits	6	5	5	5	5	6

The opcode field and funct fields together encode that the operation is addition (rather than some other arithmetic or logical operation). The three register fields of the instruction specify which registers hold the operands and which register should receive the output of the ALU. Thus, the op code and funct fields are used to *control* what data gets puts on what lines and when data gets written. I will say more about how the control signals are determined next lecture. For now, we will concentrate on the datapaths, as shown in the figure below.



Here is what happens during an `add` instruction:

- new PC value is computed and written (at the end of clock cycle)
- instruction is read ("fetched") from Memory (details later in the course)
- two ReadReg and the WriteReg are selected (recall lecture 6)
- control signals for ALU operation are determined (to be discussed next lecture)

- RegData values are read from two registers and input to ALU; ALU operation is performed
- result is written into WriteReg (at the end of clock cycle)

The steps above can all be done in a single clock cycle, provided the clock interval is long enough that all circuits have stabilized (and they would need to designed to ensure this). For example, the ALU involves carry values which take time to ripple through.

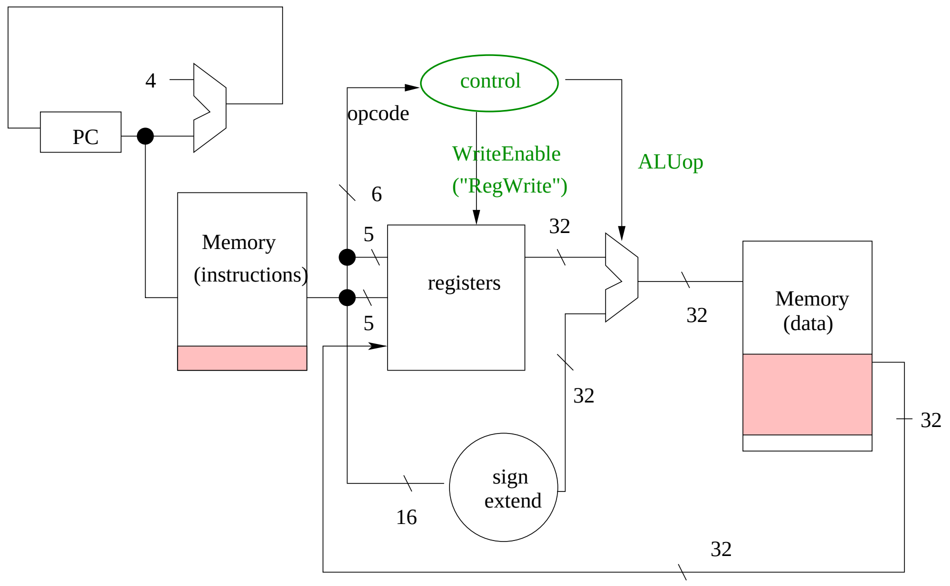
Data path for load word (`lw`)

Recall that this instruction has I format:

field	op	rs	rt	immediate
numbits	6	5	5	16

An example is: `lw $s0, 24($t3)`

Here is the data path (including the PC update!)



For `lw`, the first three steps are the same as in the `add` instruction, but the remaining steps are quite different.

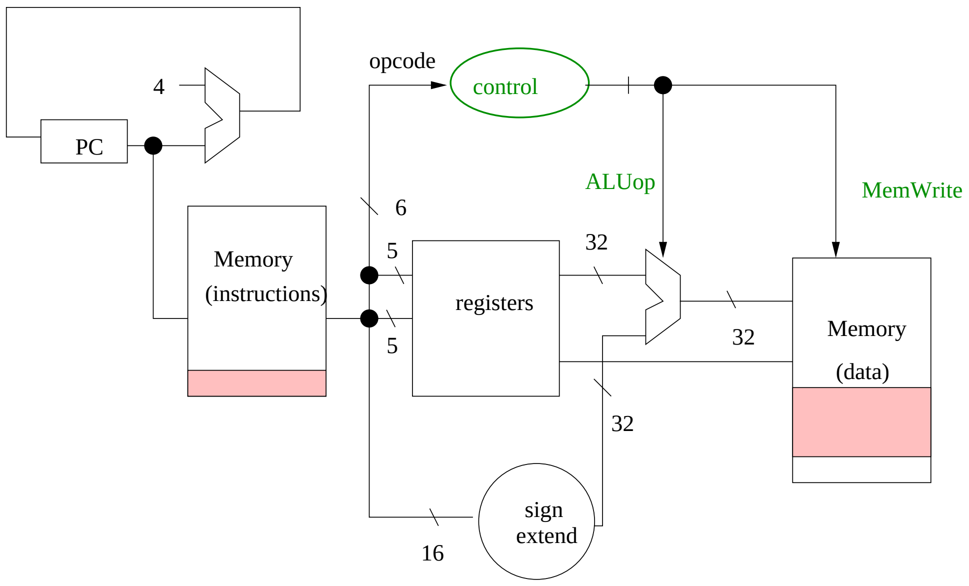
- new PC value is computed (it is written at next clock pulse)
- instruction is read ("fetched") from Memory
- ReadReg and WriteReg selected and RegWrite is specified
- base address of Memory word to be loaded is read from a register and fed to ALU
- offset (16 bit immediate field) is sign-extended and fed to ALU

- control signals are set up for ALU operation (see next lecture)
- ALU computes sum of base address and sign-extended offset; result is sent to Memory
- word is read from Memory and written into a register (end of clock cycle)

Note that in the figure above I have colored the instruction segment (left) and the data segment (right) to remind you roughly where these segments are located and the fact that they don't overlap.

Data path for store word (sw)

The format for `sw` is the same as `lw` but the data path for `sw` is slightly different:



- new PC value is computed (it is written at next clock pulse)
- instruction is read ("fetched") from Memory
- two ReadReg are selected
- base address for Memory word is read from a register and fed to ALU
- offset (16 bit immediate field) is sign-extended and fed to ALU
- control signals are set up for ALU operation
- ALU computes sum of base address and sign-extended offset; result is sent to Memory
- word is written into Memory (at end of clock cycle)

Data path for bne

Next let's look at the case that the current instruction is a conditional branch, for example,

```
bne  $s0  $s2,  label
```

which is also I format. This instruction is more interesting because the PC might not proceed to the next instruction in Memory at the next clock cycle.

To determine whether the branch condition is met, the `bne` instruction uses the ALU, and performs a subtraction on the two register arguments. If the result is not zero, then the two registers do not hold the same values and the program should take the branch. Otherwise, the program continues to the next default instruction. Specifically, if the result of the ALU subtraction is not zero, then

$$PC := PC + 4 + \text{offset}$$

else

$$PC := PC + 4 \quad .$$

It may seem a bit strange that 4 is added in both cases, but that's in fact what MIPS does. This makes sense when you consider the the datapath. The offset is relative to the current instruction address plus 4, rather than to the current instruction address. (See example on slides.)

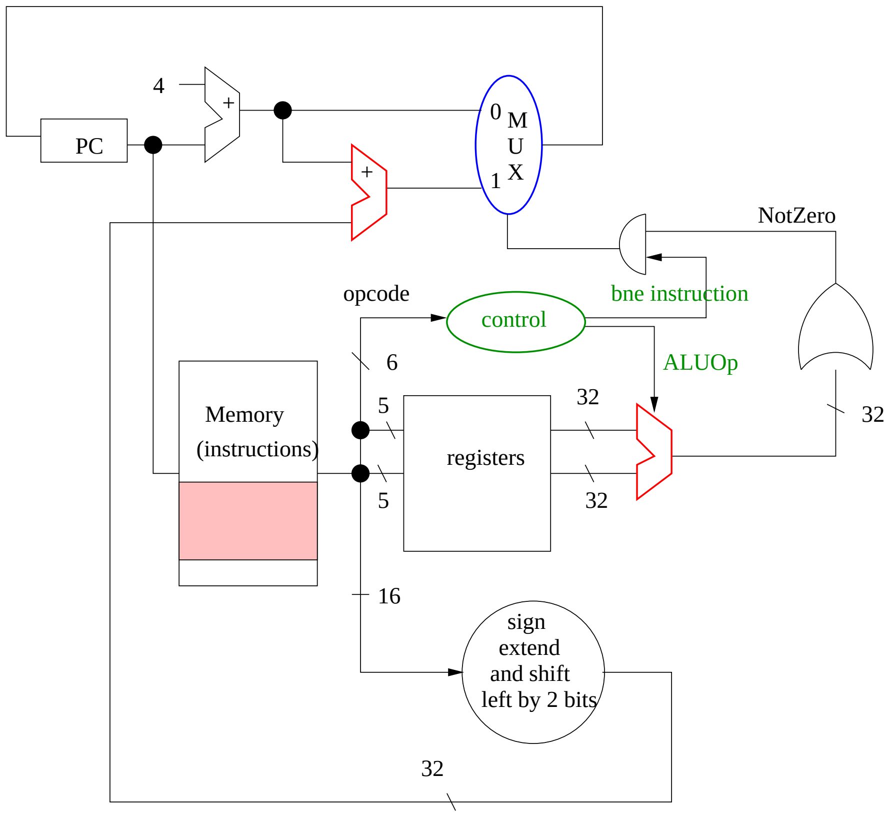
See the figure below for the datapath. The multiplexor in the upper part of the figure selects between PC+4 and PC+4+offset. The selector signal depends on the 32 bit output of the ALU, which has performed a subtraction using the two registers specified by the `bne` instruction. This 32 bit ALU result is fed into a 32 bit OR gate. The output of the OR gate is labelled "NotZero" in the figure. This NotZero signal is 1 if and only the subtraction is non-zero, that is, if the two registers do not have the same contents.

Note that the branch is only taken if indeed the instruction is `bne`. In the figure, the NotZero signal is ANDed with a bne instruction control signal that indicates it is indeed a `bne` instruction. (I have labelled the branch control as "bne instruction" in the figure.) The branch is taken when *both* the NotZero signal is ON and the branch control is ON (1). e.g. For an `add` instruction or any other R-format instruction, the branch signal would be OFF (0).

One other detail worth noting. Instructions are word aligned in Memory, namely the two lowest order bits of an instruction address always have the value 0. Thus, in the 16 bit address field of a conditional branch, MIPS doesn't waste these two bits by always setting them to 00. Instead, the 16 bit field represents an offset of 4× the offset representation in binary. That is, the offset value is "shifted" by two bits to the left before it is added to PC+4. Careful: no shift register is required for this. Instead, two 0 wires are inserted directly, and the sign extend circuit extends from 16 to 30 bits, so in total there are 32 bits.

Here are the steps of the `bne` instruction:

- PC holds the address of the current instruction
- instruction is read ("fetched") from Memory
- PC+4 value is computed
- value of PC+4 is added to sign-extended/shifted offset



- ReadReg values are set
- control signals are set up for subtraction in ALU
- RegData values are read from two registers into ALU
- ALU does subtraction and 32 bit result is fed into giant OR gate, whose output we call NotZero
- NotZero is ANDed with a bne control to select either PC+4 or PC+4+offset, and selected value is written into PC (at end of clock pulse)

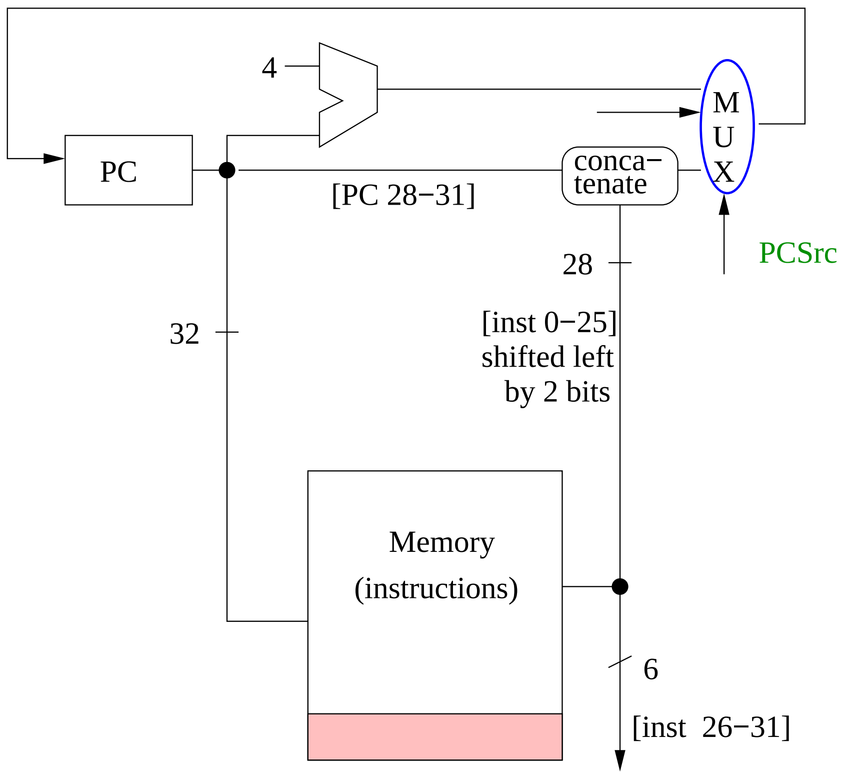
Datapath for j (jump, J format)

Recall that the jump instruction `j` has the following format. This gives 26 bits to say where we are jumping to.

field	op	address
numbits	6	26

You might think that 26 bits specify an offset from PC+4 just like the conditional branches. However, it is not done this way. Rather, the offset is from the address 0x*0000000, where $*$ is the high order 4 bits of the PC register, namely bits 28-31. Recall that the MIPS instructions of user program go up to but not including 0x10000000. Thus, all user instructions have 0000 as their highest four bits i.e. 0x0******* in hexadecimal. MIPS doesn't always concatenate 0000 onto the front of a jump instruction, since there are kernel programs as well. Instructions in the kernel programs have addresses that are above 0x80000000. So jump instructions in the kernel do not take 0000 as their upper four bits. Rather, they take $(1000)_2$ as their upper four bits, i.e. `0x8`.

Moreover, because instructions are word aligned, the 26 bit field is used to specify bits 2-27. Bits 0 and 1 in an instruction address always have the value 00.


4
M
U
X
PC
conca–
tenate
[PC 28−31]
28
PCSrc
32
[inst 0−25]
shifted left
by 2 bits
Memory
(instructions)
6
[inst 26−31]