Pipelining

MIPS instructions can be thought of as consisting of up to five stages:

- IF: instruction fetch
- ID: instruction decode and register read
- ALU: ALU execution
- MEM: data memory read or write
- WB: write result back into a register

Some instructions such as `lw` use all of these stages whereas other instructions such as `add` use just some of these stages. In a single cycle implementation, each clock cycle must be long enough for all stages to complete. This would be inefficient, however. One way to understanding this inefficiency is to note that each instruction is only in one stage at any time, and so the other stages are not doing anything during that time. The idea of pipelining is to use all stages all the time, to the extent possible. In a pipeline, instructions move along the datapath, one stage at a time, *through all stages*. As in the single cycle model, the PC is updated at every clock cycle. This implies that the next instruction begins before the previous instruction is finished. At first glance, it is difficult to believe that this could work. And indeed there are subtle issues to be addressed to make it work.

There are many familiar examples of pipelining in the world: a car wash, an assembly line in a factory, cafeteria, etc. The key idea is that you can use resources most efficiently when all workers are busy. One design challenge is that it is generally not possible to keep everyone busy at once. But you want to design the system to come as close to that as you can. The other design challenge is that there are dependencies between workers, so that when one worker doesn't do his/her/its job, then this prevents another worker from doing its job. Have a look at a clip from Charlie Chaplin's film Modern Times for an amusing example (But MUTE the volume.) `https://www.youtube.com/watch?v=DfGs2Y5WJ14`.

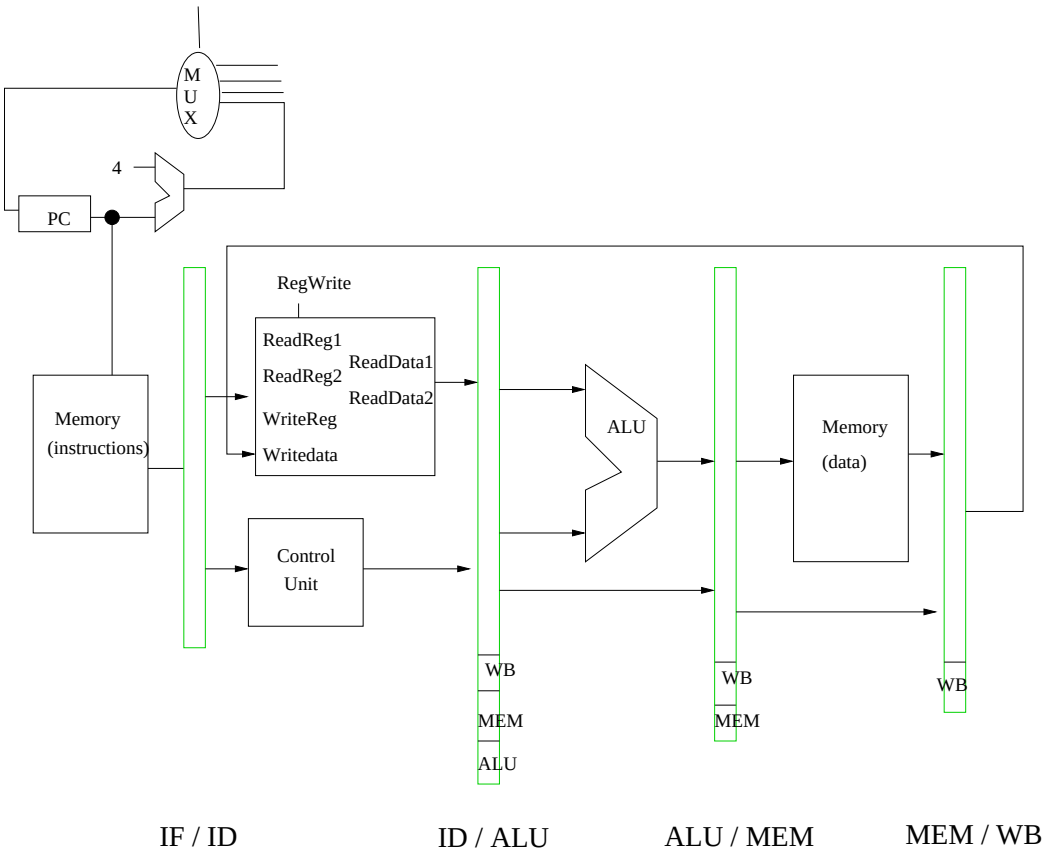
Real MIPS processors use a pipeline, and the PC is updated at every clock cycle. Each pipeline stage is given the same amount of time (one clock cycle) which is long enough that each pipeline stage can complete in this time. The five pipeline stages are listed above.

Recall that in the single cycle model, each of the control variables was given a value that depended on the instruction. (These control variables were typically either selector signals for multiplexors, or write enable signals.) The situation is more subtle in a pipeline scheme, since there are multiple instructions present in the pipeline. New registers are needed between successive stages of the pipeline, in order to keep track of which control signal is for which instruction. These registers are referred to by a pair of stages – see below. The *pipeline registers* contain all control information that is needed by that instruction, namely the values read out of the registers are the control values for an instruction, and values are written into the register at the end of the stage (end of clock cycle). These pipeline registers pass the control signals and other values through the pipeline.

- IF/ID: During a clock cycle, this register contains the instruction (one word) that was fetched from Memory in the previous clock cycle. There is no control information since the instruction

hasn't been decoded yet. At the end of the clock cycle, the instruction that is being fetched in the current clock cycle is written in this register.

- ID/ALU: This register contains parts of the fetched instruction (such as register numbers, the immediate fields, offsets) plus controls that are needed to execute the instruction. These control signals are available by the end of the ID stage, and these controls are for all remaining stages of the pipeline.
- ALU/MEM: This register contains the controls that are needed to execute the remaining two stages (MEM, WB), as well as values that were computed during the ALU stage.
- MEM/WB: This register contains controls needed to execute the WB stage, as well as any data value that needs to be written back. These data values might have been retrieved from Memory during the MEM stage, or they may be values that were computed in the ALU stage.



Many subtle issues arise in pipelining, since several instructions are processed during the same clock cycle. We are used to thinking of a program as completing one instruction before another begins, and we judge the correctness of a program with this assumption in mind. But pipelining does not obey this assumption. The rest of this lecture will address some of the problems that arise and how to solve them.

Data Hazards

Example 1

Take the two instructions:

```
add  $t1,  $s5,  $s2
sub  $s1,  $t1   $s3
```

For a single cycle machine, there is no problem with the fact that the register written to by `add` is the same as one of the registers that `sub` reads from. But a pipelined machine does have problems here. To see why, let's visualize the pipeline stages of a sequence of instructions using a diagram as follows:

clock cycles →

`add`	IF	ID	ALU	MEM	**WB**	
`sub`		IF	ID	**ALU**	MEM	WB

Notice that the `sub` instruction has executed the ALU stage *before* the `add` instruction has written its result back into the register. This will produce the wrong answer since one of the arguments of the `sub` instruction is the result of the `add`. Such a problem is called a *data hazard*. Data hazards arise when the lead instruction writes to a register that the trailing instruction is supposed to read from.

There are a few ways to avoid this problem. The easiest is to insert an instruction between the `add` and `sub` which does nothing i.e. it does not write into any of the 32 registers, nor does it write into data memory. Such an instruction is called `nop`, which stands for "no operation". This instruction is a real MIPS instruction. The key property of `nop` is that it doesn't write to any register or to memory and so it has no effect. Including `nop` instructions in a program is called *stalling*. The `nop` instruction itself is sometimes called a *bubble*.

`add`	IF	ID	ALU	MEM	**WB**			
`nop`		IF	ID	ALU	MEM	WB		
`nop`			IF	ID	ALU	MEM	WB	
`sub`				IF	ID	**ALU**	MEM	WB

Of course, this is not such a satisfying solution as the inserted `nop` instructions slow down the pipeline, which defeats the purpose of pipelining!

An alternative solution is to use the fact that the result of the `add` instruction is available at the end of the ALU stage of the `add` instruction, which is before the ALU stage of the `sub` instruction. Specifically, the result of `add`'s ALU stage is written into the ALU/MEM register, and so the result is available there. A method called *data forwarding* can be used to send the result along a dedicated path from the ALU/MEM register to an ALU input to be used in `sub`'s ALU stage.

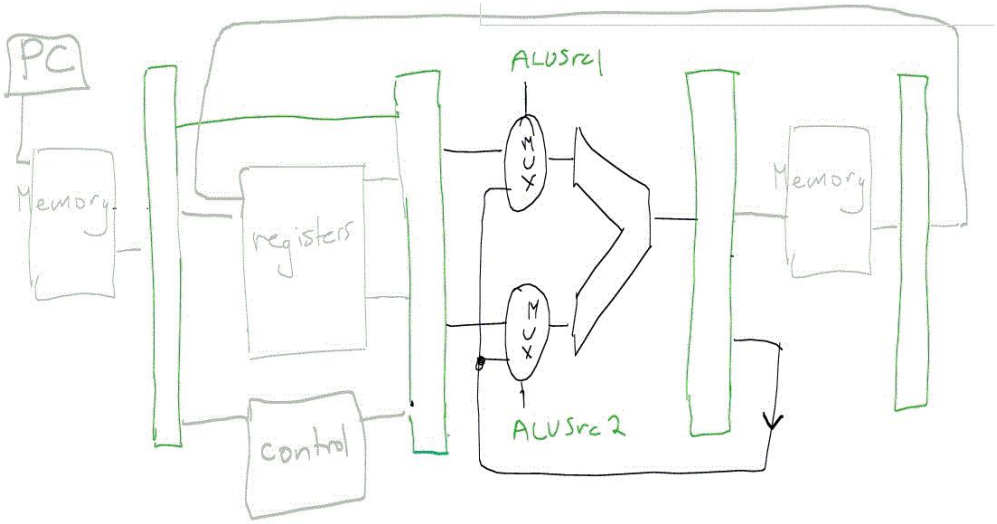
How could data forwarding be implemented? For the example above, consider the condition that could be used for data forwarding. Suppose the `add` instruction has finished its ALU stage

and (eventually) would write the result into some *rd* register. The next instruction, which would have just finished its ID stage, would use that same register (specified in its *rs* or *rt* field) as one its ALU inputs. This could be detected by checking the two conditions, which correspond to the two possible ALU inputs:

```
ALU/MEM.RegWrite & (ALU/MEM.rd == ID/ALU.rs)
ALU/MEM.RegWrite & (ALU/MEM.rd == ID/ALU.rt)
```

that is, the ALU/MEM register indicates that the leading instruction will write the ALU result into a register, and this write register is a source register used by the trailing instruction whose controls are in ID/ALU register.

To forward the data to the ALU, we could put a new multiplexor in front of each ALU input arm. Each of these would either select the usual line for its input or it would select a forwarded line (in the case that the condition above is met). I have labelled these controls ALUsrc1 and ALUsrc2.



Example 2

Here is a slightly different example.

```
lw   $s1, 24( $s0 )
add  $t0, $s1, $s2
```

	clock cycles →					
`lw`	IF	ID	ALU	MEM	**WB**	
`add`		IF	ID	**ALU**	MEM	WB

The difficulty here is that `$s1` argument for the `add` instruction has not been properly updated by the `lw` instruction until after the WB of the `lw`. You can imagine that this is quite a common hazard. The reason we load a word into a register is that we want to perform an operation on it!

The easiest (but also least effective) way to correct this hazard is to stall i.e. insert two `nop` instructions. but again this defeats the purpose of pipelining.

`lw`	IF	ID	ALU	MEM	**WB**			
`nop`		IF	ID	ALU	MEM	WB		
`nop`			IF	ID	ALU	MEM	WB	
`add`				IF	ID	**ALU**	MEM	WB

[ASIDE: Added April 26]

In fact we need three `noop` instructions here, not two. Why? When `lw` is in its WB stage, it copies a value from the MEM/WB register into the register array, but this value not yet copied into the ID/ALU register. Rather, any register values that are copied from the register array into the ID/ALU register are part of a 'later' instruction which is in its ID stage. Note that both WB and ID stages of two different instructions are using the register array at the same time. This is fine since one is writing to the register array (WB) and the other is reading from the register array (ID).

If this is confusing to you, then recall from way back in the course how a flip flop works. It consists of two D-latches, and when are writing into the first D-latch, we cannot read that value from the output of the second D latch yet. This is how we can read and write to a flipflop at the same time. What's true for D flip flops is also true for registers, since registers are made out of D flipflops.]

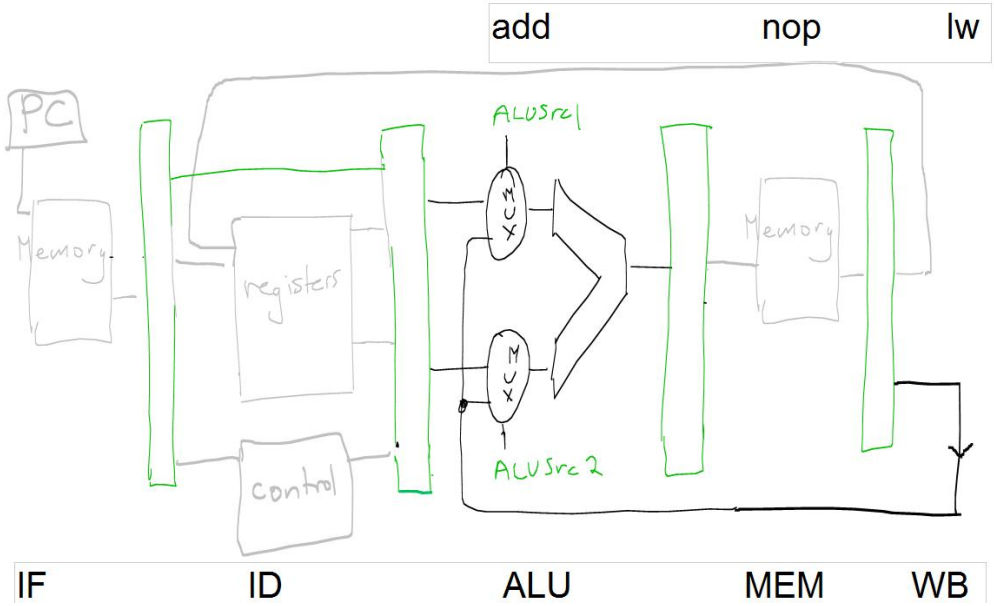
A better way to solve the problem is by data forwarding. The new value of `$s1` is written into the MEM/WB register at the end the `lw` MEM stage. So, if we stall the `add` by just one cycle, then we could forward the value read from the MEM/WB register directly into the ALU as part of `add`'s ALU stage.

`lw`	IF	ID	ALU	**MEM**	WB		
`nop`		IF	ID	ALU	MEM	WB	
`add`			IF	ID	**ALU**	MEM	WB

How would we control this? We check if the instruction in MEM/WB is supposed to write a value from Memory into a register *rd* at the end of its WB stage (namely it is `lw`), and we check if this is one of the registers that the ID/ALU instruction is supposed to read from in the ALU stage. (Note that this is two instructions behind, not one.) The conditions to be detected are either of the following:

```
MEM/WB.RegWrite & (MEM/WB.rd == ID/ALU.rs)
MEM/WB.RegWrite & (MEM/WB.rd == ID/ALU.rt)
```

When one of these conditions is met, the value that is read from Memory by `lw` can indeed be forwarded. Note that it can be forwarded to an input branch of the ALU. We could add multiplexors ALUSrc1 and ALUSrc2, to select whether this value is read to either (or both) of the input arms to the ALU.



Note that I have used the same controls ALUsrc1 and ALUsrc2 as in the previous example. Of course this will not work since these variable cannot each mean two different things. But I assume you appreciate the spirit of what we are doing here – isolating only datapaths that pertain to a particular instruction or concept. The real circuit with everything in it is enough spagetti "to feed an army".

Example 3

Another solution is to take any instructions that comes before the `lw` or after the `add` – and that are independent of `lw` and `add` instructions – and *reorder* the instructions in the program by inserting these instructions between the `lw` and `add`.

[April 13: modified]

```
sub  $t3, $t2, $s0                    lw   $s1, 40( $s0 )
lw   $s1, 24( $s0 )        ----->     sub  $t3, $t2, $s0
add  $t0, $0 , $s2                    or   $s5, $t5, $s0
or   $s5, $t5, $s0                    add  $t0, $0, $s1
```

`lw`	IF	ID	ALU	MEM	**WB**			
`sub`		IF	ID	ALU	MEM	WB		
`or`			IF	ID	ALU	MEM	WB	
`add`				IF	ID	**ALU**	MEM	WB

Control Hazards

In a single cycle implementation, the next instruction is determined by the `PCsrc` control, which is determined during the single clock cycle. In a pipeline implementation, however, `PCsrc` control is not determined right away. Rather, in the best case, it is determined during the instruction decode.

In a pipeline implementation, PC is "incremented" to PC+4 by default at the end of the instruction fetch (IF) stage, and thus the next instruction in the program enters the pipeline. However, in the case of an unconditional branch such as `j`, `jal`, `jr`, the program will not execute the instruction that directly follows the jump (unless the program explicitly jumps to that instruction, which would be unlikely). This creates a problem, since the instruction directly following the jump (at PC+4) will enter the pipeine by default.

There are two steps needed to handle jumps. First, the PC is updated at the end of the jump instructions's ID stage. That is, once the instruction is decoded, it has been determined that it is a jump instruction and it has been determined where the program should jump to, and this jump address must be written into the PC. I will not discuss this solution, since it is essentially the same as in the single cycle implementation.

Second, the instruction in the program that followed the jump and entered the pipeline should not be executed. This problem is more subtle. Consider an example:

```
                j       Label1
Label2:         addi    $s0, $s1, 0
                sub     $s2, $t0, $t1
                 :
Label1:         or      $s3, $s0, $s1
```

`j`	IF	ID	ALU	MEM	WB				
`addi`		IF	ID	ALU	MEM	WB			
`sub`			IF	ID	ALU	MEM	WB		

Again, the problem is that we do not want the `addi` (or `sub`) instruction to be executed after the `j` instruction.[1] How can we solve this problem? Inserting a nop between `j` and `addi` will work, because `addi` will not be fetched in that case. The `nop` will enter the pipeline, but the instruction following the nop will be the one that the program jumps to (after the jump's ID stage). This `nop` could be inserted by the assembler after each `j` instruction.

Inserting a `nop` instruction after each `j` is somewhat unsatisfying, however, since its effect is just to stall. Is there a better approach? Yes! Rather than inserting a `nop` after `j`, one can modify

[1]Note that the instructions following the jump might be executed *at some other time*, namely if some other part of the code jumps to `Label2`.

the opcode of the instruction that has just been fetched, namely if the opcode field in the IF/ID register is `j`, then the new value of the opcode field that is written in the next clock cycle should be the opcode of `nop`. In the above example, this would *replace* `addi` with `nop` which avoids the stall.

Example 4

For a conditional branch instruction, it is not known until the end of the ALU stage whether the branch is taken and (if so) what the address of the next instruction should be. In particular, it is not known whether the instruction at address PC+4 (following the `beq` instruction) should be executed. For example,

```
beq  $s1, $s4,  label
add  $s5, $s2,  $s0
```

`beq`	IF	ID	ALU	MEM	WB	
`add`		IF	ID	ALU	MEM	WB

We could stall by insert a sufficient number of `nop` instructions between the `beq` and `add` instructions so that, at the completion of the ALU stage of `beq`, it is known whether the branch condition is met and so at the end of that stage, the PC can be updated accordingly with the branch address or with PC+4. That is, the control *PCsrc* could be computed at the end of the `beq`'s ALU stage. In this case, *if* the branch is taken, then `add` never enters the pipeline.

`beq`	IF	ID	**ALU**	MEM	WB			
`nop`		IF	ID	ALU	MEM	WB		
`nop`			IF	ID	ALU	MEM	WB	
`add`				**IF**	ID	ALU	MEM	WB

A better solution is to put the `add` instruction into the pipeline in the default way, and then modify the `add` instruction after the ALU stage of the `beq` instruction *if the branch is taken*. For example, the `nop` opcode could be written instead of the `add` opcode into the ALU/MEM register, and all the controls could be changed to those of the `nop`. Alternatively, note that the only damage that the `add` instruction could cause is at its WB stage, where it writes the result back into a register. To prevent this damage, it would be sufficient to change the RegWrite control to 0, in the case that the branch is taken.

Notice that you certainly don't want to write assembly code which considers all of these issues. Debugging assembly code with the single cycle model in mind is difficult enough! Rather, you will to write your programs in a high level language and have a smart compiler create the assembly/machine code for you, and this compiler will know all the possible hazards and will have rules for reordering an inserting nops.