Pipelining

Pipelining[1] is a different method of multicycle implementation than the microinstruction method discussed last lecture. The goal of pipelining is to reduce the number of stages that are idle in the datapath. In a pipeline, instructions move along the datapath, one stage at a time, *through all stages.* The PC is updated at every clock cycle. Note that the next instruction begins long before the previous instruction is finished. This creates problems (known as "hazards") which will be described below.

To keep the pipeline moving along at a uniform speed, each pipeline stage is given the same amount of time (one clock cycle). This is made to be long enough that each pipeline stage can complete in this time.

In a maximally efficient pipeline, one instruction finishes at each clock cycle. To see how this can speed up performance is that, suppose it took time T_s (worst case) for an instruction to complete in a single cycle model. Suppose in a pipelined implementation, the clock cycle was $T_p < T_s$. (For 5 stages, we would like $T_p = \frac{T_s}{5}$ but that would only be possible if each stage required the same amount of time. Typically that's not the case and so $T_s > T_p > \frac{T_s}{5}$.) Thus in a maximally efficient pipeline with 5 stages, you can get a speedup of at most a factor 5. (But even a factor 2 speedup in practice would be a huge gain!)

MIPS pipeline: 5 stages

The MIPS pipeline has 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

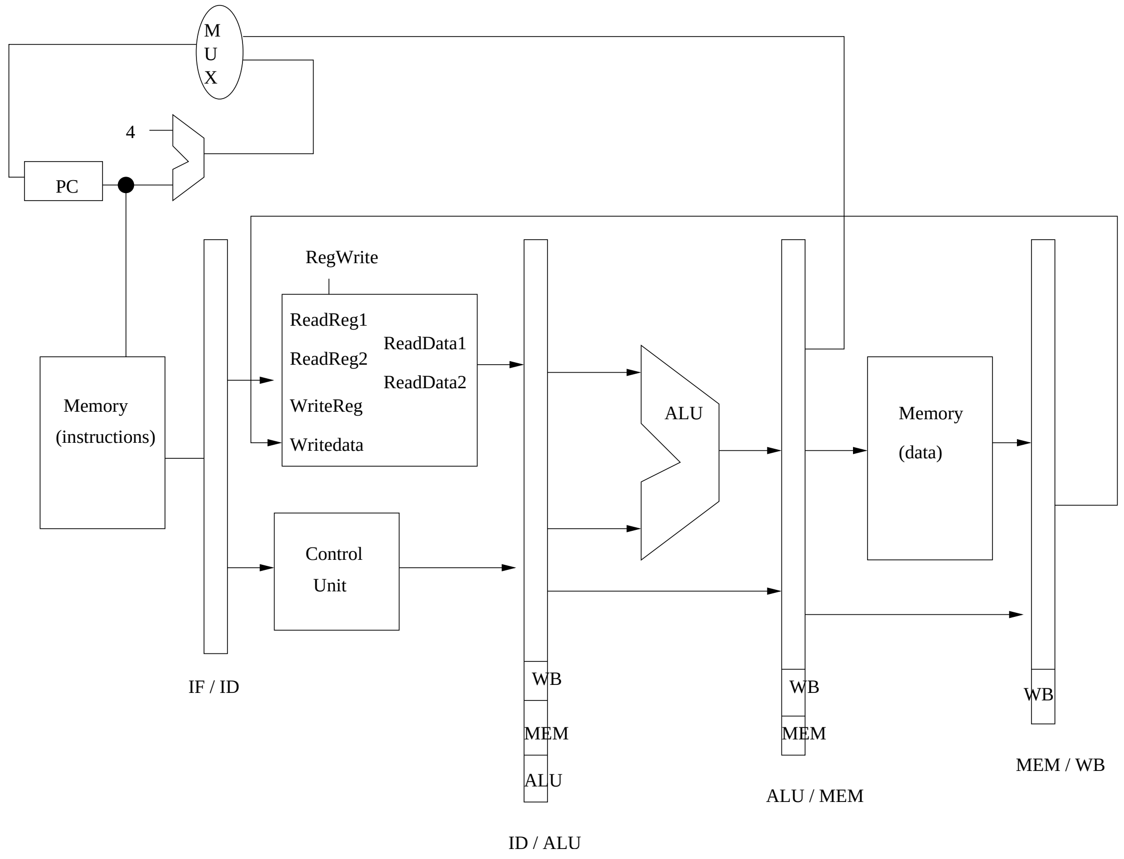
Because different instructions are present at different stages of the pipeline at any one clock cycle, new registers are needed between each successive stages of the pipeline. These *pipeline registers* contain all control information that is needed by that instruction. These registers are referred to by the pair of stages:

- IF/ID
- ID/ALU
- ALU/MEM
- MEM/WB

[1]There are many familiar examples of "pipelining" in the world: a car wash, a cafeteria, a factory assembly line. For the latter, have a look at a clip from Charlie Chaplin's Modern Times. But MUTE the volume. It is a silent film. `http://www.youtube.com/watch?v=CReDRHDYhk8`

The pipeline registers pass the control signals and other values through the pipeline. Recall the single cycle model. In that 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 IF/ID pipeline register contains the fetched instruction (one word). The ID/ALU contains the fetched instruction, plus controls that are needed to execute the instruction. (These control signals are available by the end of the ID stage and can be written into the pipeline register there - namely the register ID/ALU.) ALU/MEM register contains the controls that are needed to execute the remaining stages of the instruction (MEM, WB), as well as values that were computed in the ALU stage. MEM/WB contains controls needed to execute the WB stage, as well as any data values that need to be written back.



The pipeline registers only hold their values for one clock cycle. The values get carried along from one one pipeline register to the next at each clock cycle. The reason is simple – the registers contain all the information about an instruction being executed and each instruction gets carried through each of the five stages of the path.

Many subtle issues arise in pipelining. Several instructions are being processed during the same clock cycle but the instructions are dependent on each other. We are used to thinking of a program as completing one instruction before another begins. 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  $s1,  $s5,  $s2
sub  $t1,  $s1   $s3
```

For a single cycle machine, there is no problem with the fact that the register we are writing to in `add` is the same as one of the registers that `sub` reads from. Nor is there a problem with a multicycle model that uses microinstructions. However, 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 a satisfying solution as the inserted `nop` instructions show down the pipeline, which defeats the purpose of pipelining!

A second 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. 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 conditions that we could detect for data forwarding. An instruction (`add`) would have 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 (*rs* or *rt*) 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 instruction there will write the ALU result into a register, and this write register is also needed by the following 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. 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). *See figure in slides.*

Example 2

Here is a slightly different example.

```
lw   $s0, 40( $s1 )
add  $s3, $s1, $s0
```

clock cycles →

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

The difficulty here is that `$s0` argument for the `add` instruction has not been written into `$s0` 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 three `nop` instructions, but again this defeats the point of pipelining.

`lw`	IF	ID	ALU	MEM	**WB**				
`nop`		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 way to solve the problem is by data forwarding. The new value of `$s0` 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 condition to be detected is:

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

When this condition 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. *(Again, see the figure in the slides.)*

[ASIDE: Be aware the situation is more complicated that I just described. There are further checks to be done, *e.g.* if there is a different instruction than `nop` between the `lw` and `add` then we need to consider possible interactions with this instruction as well. In general, there are combinations of all the hazards I am identifying and the processor must be designed with all combinations considered. Details are omitted, since we are only spending one lecture on this topic.]

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`.

```
sub  $t3, $t2, $s0                     lw   $s0, 40( $s1 )
lw   $s0, 40( $s1 )        ----->      sub  $t3, $t2, $s0
add  $s3, $s1, $s0                     or   $s5, $t5, $s0
or   $s5, $t5, $s0                     add  $s3, $s1, $s0
```

`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 the typical operation of the pipeline, PC is incremented to PC+4 as part of the instruction fetch (IF) stage. This is done so that the next instruction in the program can enter the pipeline. However, in the case of an unconditional branch such as `j`, `jal`, `jr`, the next instruction to enter the pipeline is supposed to be the instruction we are jumping to (not the next instruction in the program i.e. at PC+4).

There are two steps needed to handle jumps. First, the PC is updated at the end of the jumps ID stage. That is, once the instruction is decoded, we know it is a jump instruction and we know where we are jumping to, and so we need to write this jump address into the PC. Second, we must ensure that the instruction in the program that followed the jump (at address PC+4) does not get executed. How can this be done? 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		

We do not want the `addi` and `sub` instructions to be executed since we are jumping away from them. Inserting a nop between `j` and `addi` will work, because `addi` will not be fetched in that case.

An alternative is to clobber the `addi` function by replacing it with a `nop` at runtime. That is, once `j` has been decoded (during its ID stage), it is known that a jump will occur, and so the instruction following `j` (which is currently being fetched) should not be executed. So rather than writing that instruction (`addi`) into the IF/ID register, instead you clobber it and write `nop`. This has the same effect as the solution in the previous paragraph. (Both solutions stall the pipelining by one clock cycle.)

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

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
sub  $s5, $s2,  $s0
```

`beq`	IF	ID	ALU	MEM	WB	
`sub`		IF	ID	ALU	MEM	WB

We could stall by insert a sufficient number of `nop` instructions between the `beq` and `sub` 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. In this case, *if* the branch is taken, then `sub` never enters the pipeline.

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

A better solution is to put the `sub` instruction into the pipeline in the default way, and then clobber the `sub` instruction after the ALU stage of the `beq` instruction *if the branch is taken.* (By "clobber", I just mean write the `nop` opcode instead of the `sub` opcode, and change all the controls to those of the `nop`.)

Note that the only damage that the `sub` instruction could cause is at its WB stage, where it writes the result back into a register. So, to prevent this damage, all we really need to do is change the RegWrite (write enable) control to 0, in the case that the branch is taken. If this is done, then `sub` won't write its result at the WB stage, which solves the problem.

Another solution is to take an instruction that is *before* the `beq` in the original program, and move it after the `beq`. We can do this if the instruction is independent of the `beq` instruction, i.e. it does not write into any register that `beq` instruction needs.

Here is an example where this reordering would work. Suppose the original program had the sequence:

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

then we could reorder these instructions (and insert one `nop`) to get:

```
beq $s1, s$4, label
add $s2, $s1, $s0
nop
sub $s5, s2, $s0
```

This works. After the `nop`, the ALU stage of the `beq` is complete and it is known whether the branch is to be taken, i.e. what the next value of the PC will be. If the branch is taken, the new PC value can be made to take the branch address rather than PC+4, and the `sub` instruction never enters the pipeline.

Note: if you (human) read the latter sequence with a single cycle interpretation in mind, then you would have a different interpretation of what it means than if you had the pipelined solution in mind. You would think: If the branch condition is met then the `add` instruction does *not* get executed. In fact though, in the pipeline model, the `add` instruction is executed, *even if the branch condtion is met.*

Notice that you don't want to write assembly code which considers all of these issues. Rather, you want to have a smart compiler which knows all the possible hazards and has rules for reordering and inserting nops. The compiler needs to know about the particular hardware implementation of the MIPS instructions too. It needs to know about all the control signals, and data forwarding tricks being used by the hardware.

Floating point instructions and pipelining

At the end of the lecture I briefly mentioned floaing point ops. These inherently require more than one clock cycle at the "execute" stage of the pipeline, which is the stage where the computation is done e.g. addition, subtraction, multiplication, division. (This was the ALU stage in the instructions above.) The IF, ID, MEM, and WB stages are still present for the floating point ops, but now the ALU stage is replaced by a sequence of stages e.g. FPMULT1, FPMULT2, ... FPMULT6. I did not provide any details on hazards that arise, but left it to your imagination that the hazard issues become more complicated because each execute stage requires multiple cycles.