Pipelining

February 11 2004

pipelining

when we compared the single cycle cpu to the multiple cycle cpu, we saw a decrease in cycle time, and an increase in cpi. with a pipelined processor, we get the best of both: the low cpi of the single cycle cpu, and the low cycle time of the multiple cycle cpu. the cost, as usual, is increased complexity.

note that pipelining does not improve the performance of a *single* instruction. rather, it improves the performance of groups of instructions. this is one of the classic issues in computer science: latency vs. throughput. a pipelined processor can't execute a single load instruction any faster than a multi-cycle cpu. however, a pipelined processor can execute five load instructions faster than a multi-cycle cpu.

pipelined datapath

as you look over the pipelined datapath, you should notice that it looks like a weird combination of the single and multiple cycle cpus. like the single cycle cpu, we have extra adders to compute pc+4 and the branch target. we also have separate instruction and data memories. on the other hand, we have lots of extra registers, like the multiple cycle cpu.

you'll also notice that the pipeline registers are placed in familiar locations, from our multicycle cpu. the if/id register is placed where the ir [instruction register] register was, the id/ex register is placed where a and b were, the ex/mem register is placed where aluout was, and the mem/wb register is placed where mdr [memory data register] was. this results in the same five stages that we know so well from our multi cycle cpu: fetch, decode, execute, memory, writeback.

unlike the multi cycle cpu, the pipelined datapath requires that every instruction use all five stages of execution. why? because we end up with resource conflicts if instructions with different latencies are run in parallel.

suppose we use the latencies from our multi-cycle cpu, and we try to run a load instruction followed by an add instruction. the load instruction will require five cycles to execute, and the add instruction will require four cycles. so, if we start running the load instruction on cycle 1, it will finish execution on cycle 5. we are pipelining, so we can start running the add instruction on cycle 2, and it will finish on cycle 5. this is a problem: we have two instructions finishing on cycle 5: they will both try to write to the register file on cycle 5. this is a problem, because our register file only has one write port.

we eliminate these types of resource conflicts by requiring every instruction to go through all five cycles of execution.

pipelined control

control for a pipelined processor is quite similar to the control for the single cycle cpu: it's just combinational logic. we can do this because as soon as we know what type of instruction we're executing, we know how the control signals need to be set for each stage of its execution.

the trick is to generate *all* the control signals for each instruction during decode, and to push the values of the control signals through the pipeline. this ensures that each instruction knows how its control signals need to be set for each stage of its execution.

pipeline timing diagrams

your typical pipeline timing diagram looks something like this [f = fetch, d = decode, x = execute, m = memory, w = writeback]:

instruction	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8	cycle 9
add $1, $2, $3	f	d	x	m	w				
sub $4, $10, $20		f	d	x	m	w			
and $6, $16, $11			f	d	x	m	w		
lw $8, 4($11)				f	d	x	m	w	
or $10, $18, $11					f	d	x	m	w

these diagrams carry a whole lot of information... this can be a bit confusing, so let's make sure we understand what it means. each row of the diagram indicates when each instruction goes through each pipeline stage. so, if we read across the row marked "sub $4, $10, $20", we see that the subtract instruction is fetched in cycle 2, it decodes in cycle 3, it executes in cycle 4, etc.

each column of the diagram shows which pipeline stages are in use at any instant in time, and it tells us which instructions are using those pipeline stages. so, as we read down the column marked "cycle 3", we know that in the third cycle, the add instruction is in the execute stage, the subtract instruction is in the decode stage, and the "and" instruction is in the fetch stage.

data hazards

data hazards occur when our processor tries to read data before the data has been written. note that all data hazards can be resolved by *waiting*. we didn't have to worry about data hazards in our multiple cycle cpu - and we can turn our

pipelined processor into a multiple cycle cpu by inserting four useless instructions ["no-ops"] in between every pair of adjacent instructions.

let's look at the following code sequence:

```
add $1,  $2, $3
sub $4,  $1, $2
and $6,  $6, $1
lw  $8,  4($1)
or  $10, $8, $1
```

let's find the data hazards. the easiest way to do this is to draw a pipeline timing diagram showing all the instructions moving through the pipeline normally:

instruction	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8	cycle 9
add $1, $2, $3	f	d	x	m	w				
sub $4, $1, $2		f	d	x	m	w			
and $6, $6, $1			f	d	x	m	w		
lw $8, 4($1)				f	d	x	m	w	
or $10, $8, $1					f	d	x	m	w

now we look for trouble.

1. the add instruction will write $1 in cycle 5, but the sub instruction will read $1 in cycle 3.
2. the add instruction will write $1 in cycle 5, but the "and" instruction will read $1 in cycle 4.
3. the lw instruction will write $8 in cycle 8, but the "or" instruction will read $8 in cycle 6.

what can we do about these problems? the first two problems can be resolved by *forwarding*.

let's look at the first problem. the add instruction won't write the value of $1 into the register file until cycle 5, but the add instruction actually computes the value of $1 in cycle 3, when it actually does the addition. and the sub instruction doesn't *really* need the value of $1 until cycle 4, when it actually does the subtraction. so, if we can *forward* the value computed by the add instruction in cycle 3 to the sub instruction in cycle 4, we can eliminate this hazard.

similarly, the "and" instruction doesn't really need the value of $1 until cycle 5, when it actually does the "and" operation. so if we can forward the value computed by the add instruction in cycle 3 to the "and" instruction in cycle 5, we can eliminate this hazard.

the third problem is a bit trickier. the data read from memory by the lw instruction won't be available until the end of cycle 7, at the earliest. but the "or" instruction needs this data at the beginning of cycle 7. there's no way to resolve this

hazard by forwarding... the "or" instruction must wait one cycle for the data to be read from memory. this results in a *pipeline stall*.

forwarding

we always forward data from the pipeline register at the *beginning* of one stage to be used at the *beginning* of another stage. this is very important!

it should be fairly obvious why we always forward *to* the beginning of another stage: we forward values when they are needed, and values are always needed at the beginning of pipeline stages.

so why do we always forward *from* the pipeline register at the beginning of the source stage? consider the effects of forwarding from the end of one stage to the beginning of another. the length of the critical path doubles. the critical path now goes like this: all the way through the source stage, through the forwarding logic, and through the target stage. since our critical path is now at least twice as long as it was before, our cycle time doubles. this is bad.

so to summarize: we always forward data from the pipeline register at the *beginning* of one stage to be used at the *beginning* of another stage. :)

let's look at how forwarding can resolve the first two problems described above.

the add instruction will compute the new value of $1 in cycle 3, and write it into the ex/mem pipeline register at the end of cycle 3. at the beginning of cycle 4, we need to forward the value from the ex/mem pipeline register into the the alu, where it will be used by the sub instruction.

the "and" instruction needs the value of $1 at the beginning of cycle 5. the add instruction computed the value of $1 way back in cycle 3, and now the value of $1 is just moving through the pipeline towards the writeback stage. so in cycle 5, we can grab the value of $1 from the mem/wb pipeline register, and forward it to the alu, where it will be used by the "and" instruction.

stalls

we saw above that we can't remove all our data hazards through forwarding - sometimes we just have to wait. these are pipeline stalls.

we saw in our example above that the lw instruction will read the new value of $8 from memory in cycle 7, but the "or" instruction needs this value of $8 at the beginning of cycle 7. if you're thinking "why can't we forward the value of $8 from the end of the memory stage to the beginning of the execute stage?", re-read the section above on forwarding, and why we can't forward from the end of one stage to the beginning of another. :)

if we make the "or" instruction wait one cycle, it will execute in cycle 8, and it can use our existing forwarding logic to grab the value of $8 from the mem/wb register and forward it into the alu.

before we think about how to make the "or" instruction wait one cycle, we need to think about *when* we want the "or" instruction to wait one cycle. we'd like the stall to occur *as early as possible*, because it reduces the amount of work our hazard unit needs to do. in our example, we know that we will have to stall the processor in cycle 6, when the "or" instruction decodes. in cycle 6, we know that the "or" instruction reads $8, and we know that the lw instruction will write $8. we need to make the "or" instruction wait one cycle.

to summarize: we always introduce stalls *as early as possible*. this means that we stall the processor when we *decode* an instruction that uses the value read from memory by a preceding load instruction.

so how do we make the "or" instruction wait one cycle? the first thing to do is to replace the "or" instruction with a no-op. to do this, we can just write zeroes into the id/ex pipeline register. by zeroing out the id/ex pipeline register, we effectively insert a bubble into the pipeline: since all the control signals will be zero, regwrite=0, memread=0, memwrite=0, etc, so nothing will happen as the zeroes pass through each remaining pipeline stage.

next, we need to make the "or" instruction re-decode on the next cycle. similarly, the instruction waiting in line behind the "or" instruction needs to re-fetch on the next cycle. to accomplish these two tasks, we can disable writes on the id/ex pipeline register, and disable writes to the pc.

let's take a look at what our pipeline timing diagram looks like after we're done:

instruction	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6	cycle 7	cycle 8	cycle 9	cycle 10
add $1, $2, $3	f	d	x	[a] m	[c] w					
sub $4, $1, $2		f	d	[b] x	m	w				
and $6, $6, $1			f	d	[d] x	m	w			
lw $8, 4($1)				f	d	x	m	[e] w		
or $10, $8, $1					f	s	d	[f] x	m	w

there are forwarding arrows from [a] to [b], [c] to [d], and [e] to [f]. stupid html not letting me draw arrows... the "s" in cycle 6 is a pipeline bubble.

problem

draw a pipeline timing diagram that shows how the following code executes through the pipeline. show any forwarding that must take place, and any stalls that must be introduced.

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```