

Final Project

Alex Clemmer

Student number: u0458675

1 Analysis of Peter's Design:

1.1 Analysis of Clock Period:

The clock period is 2 ns + the clock penalty. Given the penalty definition:

$$\frac{13 \text{ instructions}}{8 \text{ ns}} = 1.625 \text{ ns} \quad (1)$$

Rounding down, this gives us a clock period of **2 + 1 ns = 3 ns**.

1.2 Latencies in cache and I/O:

We want to make the common case fast. In the case of fetching **I-Mem** to **IR**, we want the actual memory access time to be incorporated into the cycle itself. Why? Because it is silly to waste a stall cycle literally every time you fetch an instruction. Besides that, fetching from cache only costs 1 ns, which is very probably small enough to be incorporated the cycle that requests the memory.

Since the common case of fetching data memory is not fundamentally different, and because data access is so heavily used, it makes sense that this reasoning should hold for **D-Mem** accesses as well.

Cache misses are not that much more complicated: we miss when looking in the cache, and then incur a 1 ns of wait time for the memory controller followed immediately by 10 ns of wait time for the memory access. This gives us 4 cycles of latency.

The spec for the assigned machine says that a miss-to-I/O ends up going through cache to the memory controller and then to I/O. The memory controller costs 1 ns. The spec also says that, "once activated", the I/O will supply the message in 5 ns. So I/O read will cost 1 + 5 ns, or 6 ns, which is 2 full cycles.

The write-I/O should work the same way. It should go from cache miss to memory controller to I/O write, which would take 5 ns. Once again, this would give us 2 full cycles.

Waiting for device writes is, yes, crucial. If we didn't, if instead, we simply gave it 1 cycle of latency, we would not get to actually write all the data. Important to note is that the whole word is written or read at the end of the 5 ns, though, so it is not necessary to wait beyond the decided 6 ns total for these things to complete.

1.3 Instruction Analysis:

Important notes and extended discussion:

Much of the processor's work is transparent at the level of the assembly programming. I – and programmers in general – will not have to worry about microdelays like latch times, at least not for the purposes here.

What we will have to worry about is parsing out what exactly is meant in several places in this assignment, which I try to do with a dose of pragmatism, cost balancing, intuition, and occasionally, high-level discussions with people who know a lot about that particular area. Two particular things that worried me on the outset were:

(1) Initially there was some concern about the effect of a cache hit on a memory cycle. Does it stall? Or does it fit inside the clock cycle? Most of our work has taken a pretty naive view of memory, and so it is easy to make mistakes here.

Generally we want to keep the common case fast, so I did decide that a cache hit will fit inside the cycles. The reasons I decided this are, first, that fetching from **I-Mem** should not cause a delay cycle every time (that would be ridiculous), and second, that the cache hits take 1 ns which is really manageable inside of one cycle.

(2) Are we using write-backs, or write-through? Intuitively, write-backs seems like the right answer, but this answer is not given explicitly. It is not, for our purposes, hugely important, but it is not irrelevant, either.

(3) I am assuming they get loaded into memory at device start and never get overwritten after that. In other words, I am assuming we never miss to I/O for the instructions. This probably doesn't need to be said, but I am saying it anyway.

(4) I am glomming cycles into groups that have the same possible stall patterns. For example, `loada` and `storea` can both miss for data, and also miss-goto-I/O; these things are not always possible for all instructions.

1.3.1 seta, deci, flush, tai, tbi, tia, tib:

These are instructions that have 3 segments: 1 for fetching instruction to IR, 1 for incrementing the PC, and 1 for performing the operation.

The only data hazard here is in the instruction fetch stage, which is the first stage. The other two segments are highly stable.

Status	C1	C2	C3	Total
Hit	1 cycle	1 cycle	1 cycle	3 cycles
Miss	1 cycle + 4 cycles stall	1 cycle	1 cycle	7 cycles

1.3.2 loada, loadai, storea, storeai, cmpai:

These instructions also tri-segmented, with the added twist of being able to have two misses, and, on the second one, two *types* of miss.

The first opportunity to miss, again, is the instruction fetch segment, which is the first segment. The second opportunity to miss is the third segment, and in addition, it can miss to I/O, which causes additional penalties. The middle segment is stable.

Status	C1	C2	C3	Total
H + H	1 cycle	1 cycle	1 cycle	3 cycles
M + H	1 cycle + 4 cycle stall	1 cycle	1 cycle	7 cycles
H + M	1 cycle	1 cycle	1 cycle + 4 cycle stall	7 cycles
M + M	1 cycle + 4 cycle stall	1 cycle	1 cycle + 4 cycle stall	11 cycles
H + I/OM	1 cycle	1 cycle	1 cycle + (1 ns + 5 ns)	5 cycles
M + I/OM	1 cycle + 4 cycle stall	1 cycle	1 cycle + (1 ns + 5 ns)	9 cycles





1.3.3 bge, blt:








































These instructions take 2 cycles to complete plus memory stalls. The second cycle is stable, and thus the first cycle is the only one which may miss:

Status	C1	C2	Total
Hit	1 cycle	1 cycle	2 cycle
Miss	1 cycle + 4 cycles stall	1 cycle	6 cycles

1.4 Running time of algorithm


Here's the really tricky part. The (rather large) table that contains all the information on the instructions, how many times they get executed, and so on, occurs on the next page, along with a key that shows by color how many times each statement in particular gets executed.

Key	
	executes n times
	executes $\frac{n(n-1)}{4}$ times
	executes $\frac{n(n-1)}{2}$ times
	executes 1 time

				Mem	Label	Tag	Index	Offset	Assoc. #	Instruction	IH	IM	DH	DM	I/O
				100	Start	01100	<u>10</u>	0	1	seta		IM			
				101		01100	<u>10</u>	1	1	tai	IH				
				102	InLoop	01100	<u>11</u>	0	1	flush		IM			
				103		01100	<u>11</u>	1	1	loada	IH				I/O
				104		01101	<u>00</u>	0	1	storeai		IM			
				105		01101	<u>00</u>	1	1	deci	IH				
				106		01101	<u>01</u>	0	1	bge		IM			
				107	Sort	01101	<u>01</u>	1	1	seta	IH				
				108		01101	<u>10</u>	0	2	storea		IM		DM	
				109		01101	<u>10</u>	1	2	tai	IH				
				110	Outer	01101	<u>11</u>	0	2	loadai		IM		DM	
				111		01101	<u>11</u>	1	2	storea	IH		DH		
				112		01110	<u>00</u>	0	2	tib		IM			
				113	Keep	01110	<u>00</u>	1	2	loadai	IH		DH		
				114	Inner	01110	<u>01</u>	0	2	deci		IM			
				115		01110	<u>01</u>	1	2	blt	IH				
				116		01110	<u>10</u>	0	1	cmpai		IM	DH		
				117		01110	<u>10</u>	1	1	blt	IH				
				118	Better	01110	<u>11</u>	0	1	tib		IM			
				119		01110	<u>11</u>	1	1	bge	IH				
				120	NoBetter	01111	<u>00</u>	0	1	cmpai		IM	DH		
				121		01111	<u>00</u>	1	1	bge	IH				
				122		01111	<u>01</u>	0	1	blt		IM			
				123	Swap	01111	<u>01</u>	1	1	storea	IH			DM	
				124		01111	<u>10</u>	0	2	loada		IM		DM	
				125		01111	<u>10</u>	1	2	tbi	IH				
				126		01111	<u>11</u>	0	2	storeai		IM		undef	
				127		01111	<u>11</u>	1	2	loada	IH		DH		
				128		10000	<u>00</u>	0	2	tai		IM			
				129		10000	<u>00</u>	1	2	loada	IH				
				130		10000	<u>01</u>	0	2	storeai		IM	DH		
				131		10000	<u>01</u>	1	2	deci	IH			most	
				132		10000	<u>10</u>	0	1	tia		IM			
				133		10000	<u>10</u>	1	1	storea	IH			DM	
				134		10000	<u>11</u>	0	1	bge		IM			
				135		10000	<u>11</u>	1	1	seta	IH				
				136		10001	<u>00</u>	0	1	tai		IM			
				137	OutLoop	10001	<u>00</u>	1	1	loadai	IH		$\frac{1}{2}$	$\frac{1}{2}$	
				138		10001	<u>01</u>	0	1	storea		IM			I/O
				139		10001	<u>01</u>	1	1	deci	IH				
				140		10001	<u>10</u>	0	1	bge		IM			
				141		10001	<u>10</u>	1	1	blt	IH				

Instruction analysis

The breakdown of each instruction should be as follows (organized by memory number):

 executes 1 time

These instructions are executed a total of once. For example, the first two are really for initialization purposes more than anything else, and never get looped back to.

- 100** seta IM 7 cycles, executed once.
- 101** tai IH 3 cycles, executed once.
- 107** seta IH 3 cycles, executed once.
- 108** storea IM 7 cycles, executed once.
- 109** tai IH 3 cycles, executed once.
- 135** seta IH 3 cycles, executed once.
- 136** tai IM 7 cycles, executed once.
- 141** blt IH 2 cycles, executed once.

 executes n times

Instructions 102-106 read data from I/O. They get executed 100 times in total, and after the first time, they will be IH (instruction hit). 103 is always an I/O read, and therefore will always incur this penalty. 104 is in the trick position of having every alternating data read be a hit; this is because when we cache a word, we end up pulling two into the cache, and so the next one is a hit.

- 102** flush IM 7 cycles, executed once + IH 3 cycles executed 99 times. Total **304** cycles.
- 103** loada IH I/O $5 \cdot 100 = \mathbf{500}$ cycles.
- 104** storeai 1 IM DM + 49 IH DM + 50 IH DH = 11 cycles + 343 cycles + 150 cycles = **603** cycles.
- 105** deci 100 IH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 106** bge 1 IM + 99 IH, or $6 + 99 \cdot (3) = \mathbf{303}$ cycles.

Instructions 110-112 and 123-134 are part of the “Outer” loop, which you can sort of see in the above diagram (they are the massive blocks of red, straddling the inner loop). Unlike previously, the instruction cache is not large enough to preserve all the instructions, so they get overwritten. That means we have to pay the hit cost on the instructions that miss on every single loop iteration.

It should be noted that you can actually get a data hit on 126 under very specific circumstances, but we will assume that it misses here, because it happens a lot more often.


- 110** loadai 100 IM DM, or $100 \cdot (11) = \mathbf{1100}$ cycles.
- 111** storea 100 IH DH, or $100 \cdot (11) = \mathbf{1100}$ cycles.
- 112** tib 100 IM, or $100 \cdot (7) = \mathbf{700}$ cycles.
- 123** storea 100 IH DM, or $100 \cdot (7) = \mathbf{700}$ cycles.
- 124** loada 100 IM DM, or $100 \cdot (11) = \mathbf{1100}$ cycles.
- 125** tbi 100 IH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 126** storea 100 IM DM, or $100 \cdot (11) = \mathbf{1100}$ cycles.
- 127** loada 100 IH DH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 128** tai 100 IM, or $100 \cdot (7) = \mathbf{700}$ cycles.
- 129** loada 100 IH DH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 130** storeai 100 IM DM, or $100 \cdot (11) = \mathbf{1100}$ cycles.
- 131** deci 100 IH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 132** tia 100 IH, or $100 \cdot (7) = \mathbf{700}$ cycles.
- 133** storea 100 IH DH, or $100 \cdot (3) = \mathbf{300}$ cycles.
- 134** bge 100 IM, or $100 \cdot (6) = \mathbf{600}$ cycles.

Finally, the loop that sends all the data to the I/O device should take 100 times also. Like the “input” loop above, after the first time, all the instructions will be cached, and thus, will never again incur an IM.

Also like above, data for 137 will miss half the time, because when it does miss, it will pull two words into cache, so the next one will actually hit. Also, 138 clearly is another miss-to-I/O, and since there’s no way to avoid it, this will happen every single time.

- 137** loadai 50 IH DH + 50 IH DM, or $50 \cdot (3) + 50 \cdot (7) = \mathbf{500}$ cycles.
- 138** storea 1 IM I/OM + 99 IH I/OM, or $1 \cdot (9) + 99 \cdot (5) = \mathbf{504}$ cycles.
- 139** deci 100 IH, or $100 \cdot (3) = \mathbf{300}$ cycles.

140 bge 1 IM + 99 IH, or $1 \cdot (6) + 99 \cdot (2) = \mathbf{204}$ cycles.

 executes $\frac{n(n-1)}{2}$ times

This section corresponds to the inner loop. I confess I had a hard time determining how many times the inner loop runs, but then a friend of mine tipped me off to it being an expression of the standard geometric series (which made me feel stupid for having forgotten). So it (the inner loop) ends up getting executed $\frac{100(100-1)}{2}$ or 4950 times.


Other than the fact that the inner loop has two control paths (**Better** and **NoBetter**), the execution time ends up being pretty straightforward: the first time through, all IMs take effect, but the second time through, the instruction is cached, and it ends up being a hit.

114 deci 100 IM + 4850 IH, or $100 \cdot (7) + 4850 \cdot (3) = \mathbf{15250}$ cycles.

115 blt 4950 IH, or $4950 \cdot (2) = \mathbf{9900}$ cycles.

116 cmpai 50 IM DH + 2425 IH DM + 50 IM DM + 2425 IH DH, or $50 \cdot (7) + 2425 \cdot (7) + 50 \cdot (11) + 2425 \cdot (3) = \mathbf{20300}$ cycles.

117 blt 4950 IH, or $1 \cdot (6) + 2474 \cdot (2) = \mathbf{4954}$ cycles.

 executes $\frac{n(n-1)}{4}$ times

This section corresponds to the **Better** and **NoBetter** control paths inside the inner loop. I will assume for simplicity that **Better** and **NoBetter** will be executed each half of the amount that the inner loop gets executed because, although that's not technically correct, it doesn't matter on a great scale in this case. Thus the green squares will get executed $\frac{4950}{2} = 2475$ times.

One thing that is interesting is that they are both the same amount of instructions long, so if the runtime is shorter because one is taken more often than the other, it is not *that* much shorter.

There's only one "tricky" part. We must remember that, for 100 of those, the instructions will have to be re-cached because the loop has restarted.

113 loadai 2475 IH DH, or $2475 \cdot (3) \mathbf{7425}$ cycles.

118 tib 2425 IH + 50 IM, or $2425 \cdot (3) + 50 \cdot (7) = \mathbf{7625}$ cycles.

119 bge 2475 IH, or $\mathbf{4950}$ cycles.

120 cmpai 1 IM DH + 2475 IH DH, or $7 \cdot (1) + 2475 \cdot (3) = \mathbf{7432}$ cycles.

121 bge 2475 IH, or $2475 \cdot (2) = \mathbf{4950}$ cycles.

122 blt 1 IM + 2474 IH, or $1 \cdot (6) + 2474 \cdot (2) = \mathbf{4954}$ cycles.

1.4.1 Totals

When you add all these cycles together you get **101,693** cycles in total. Each cycle lasts 3 ns, thus the total time to sort 100 16-bit integers is 305,079 ns, or 0.000305 seconds. That means we can do 3277.84 sorts in a second.

1.5 How can it be better?

1.5.1 The algorithm

The first thing to look at when optimizing is obviously the algorithm. And it turns out that this tree is ripe for the picking: most of our cycles are wasted on the inner loop, and in particular, comparing numbers. Insertion sort is $O(n^2)$, so this can almost certainly be improved. Mergesort is probably too big, quicksort is not constant, and radix might also be too big. We may not be able to implement a really, really efficient algorithm, but we can certainly do better than this. It is clear that we do not need to perform 2500 comparisons per 100 instructions.

1.5.2 Get rid of clock penalty

One third of all nanoseconds are wasted on the clock penalty. This does not need to be the case. Even if we incur heavy penalties from decreasing instructions, we will still be saving a third of all nanoseconds spent. This is clearly well worth it.

1.5.3 The cache

We spend enough time waiting on stalled data in the inner loop that it's a problem, and as we shrink the runtime, this will increase in proportion. We, first, need more of it (as much as possible), and we need to organize it better. So the solution (at least *prima facie*) would be to shrink the instructions needed to do this stuff. Part of that is designing a better algorithm, of course. As for organizing the cache, I think it would be worth it to trade the timing penalties to make it a lot more associative, but this may be up for discussion.