Department of Electronics

Computer Architectures

Homework Two

Abstract

The second homework assignment for the second year, Computer Architectures module from the Department of Electronics at the University of York.

Y3839090

May 8, 2017

Contents

1 Paged Segmentation 1

2 Caching 2
2.1 direct-mapped cache . . . 2
2.1.1 sizes . . . 2
2.1.2 hits and misses . . . 2
2.2 fully-associative cache . . . 3
2.2.1 sizes . . . 3
2.2.2 hits and misses . . . 4
2.3 set-associative cache with 2 blocks per set with a a not-last-used replacement policy . . . 4
2.3.1 sizes . . . 4
2.3.2 hits and misses . . . 4
2.4 set-associative cache with 8 blocks per set with a least-recently-used replacement . . . 5
2.4.1 sizes . . . 5
2.4.2 hits and misses . . . 6

3 Hazards 7
3.1 RAW Data Hazards Running On Arch D . . . 7
3.2 Eliminating hazards without forwarding . . . 7
3.3 Eliminating data hazards with forwarding . . . 9
3.4 Branch prediction . . . 9
3.4.1 Without any form of speculative execution . . . 9
3.4.2 With speculative execution using static prediction – predict not taken 9
3.4.3 With speculative execution using static prediction – predict taken 10
3.4.4 With speculative execution using static prediction – direction-based prediction . . . 10

List of Figures

1 A Flowchart of the algorithm for paged segmentation, assuming the presence of separate translation look aside buffers (TLBs) for pages and segments. 1

List of Tables

1 block placement in a direct-mapped cache 3
2 block placement in a set-associative cache with 2 blocks per set with a not-last-used replacement policy 5
3 block placement in a set-associative cache with 8 blocks per set with a least-recently-used replacement policy 6

1 Paged Segmentation

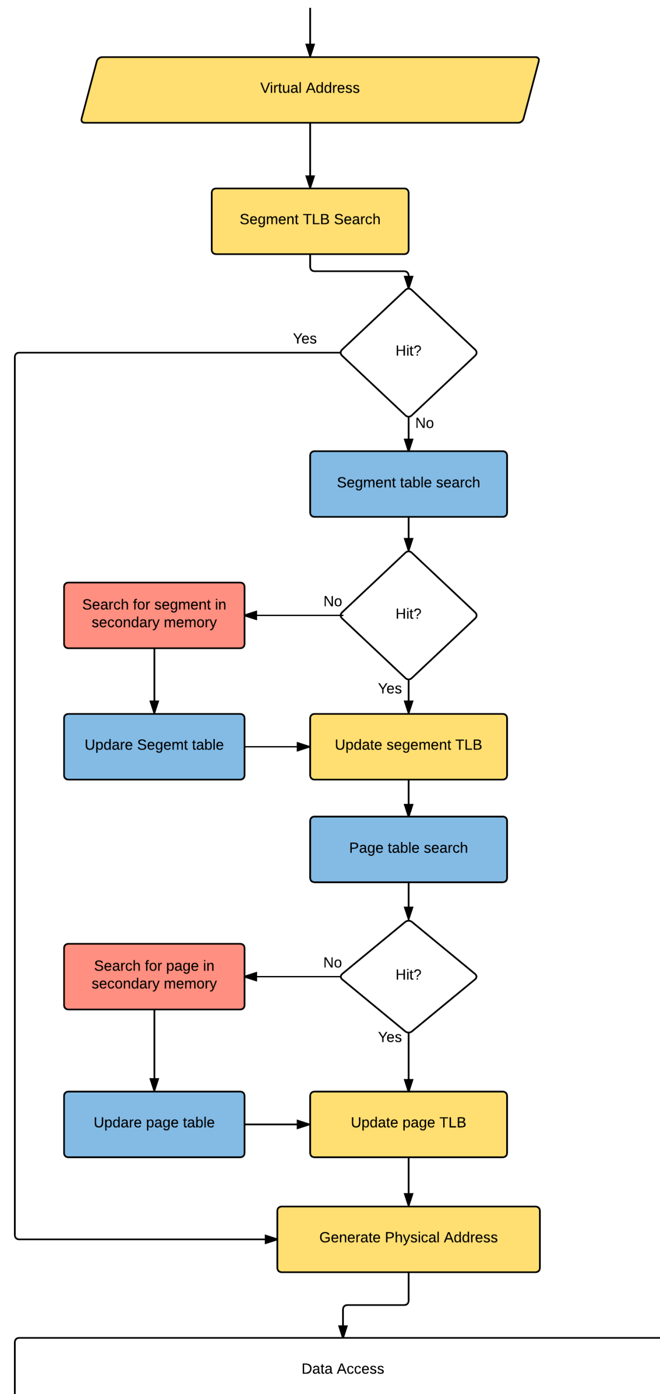

Figure 1: A Flowchart of the algorithm for paged segmentation, assuming the presence of separate translation look aside buffers (TLBs) for pages and segments.

2 Caching

$addresssize = 32b$

$blockssize = 64words = 2048b$

$wordsize = 32b$

$cachesize = 16kB = 131072b$

$numblocks = cachesize/blocksize = 64$

2.1 direct-mapped cache

2.1.1 sizes

Offset : $size(\text{offset}) = (blocksizeB/wordsizeB) + wordsizeB = (256/32) + 4 = 12b$

Index : $size(\text{index}) = log_2(cachesize/blocksize) - 1 = log_2(16kB/256B) - 1 = 8b$

Tag : $size(\text{tag}) = addresssize - size(\text{offset}) - size(\text{index}) = 32 - 12 - 8 = 12b$

2.1.2 hits and misses

I used excel to automate finding the cache block placement for each of the blocks in the three arrays. From these I was then able to manually work out the total hits and misses. Memory location are expressed in decimal because excel cannot deal with Hexadecimal values.

Table 1: block placement in a direct-mapped cache

A	Block Offset				Total	
	0	64	128	192	Hits	Misses
majority hit/miss	m	m	m	m	0	256
phys addr (byte)	2711684608	2711684864	2711685120	2711685376		
phys block (block)	10592518	10592519	10592520	10592521		
cache block (block)	6	7	8	9		

B	Block Index				Total	
	0	64	128	192	Hits	Misses
majority hit/miss	m	m	m	m	0	256
phys addr	2711750144	2711750400	2711750656	2711750912		
phys block	10592774	10592775	10592776	10592777		
cache block	6	7	8	9		

C	Block Index				Total	
	0	64	128	192	Hits	Misses
majority hit/miss	h	h	h	h	252	4
phys addr	3165496832	3165497088	3165497344	3165497600		
phys block	12365222	12365223	12365224	12365225		
cache block	38	39	40	41		

Array A and B map to the same location in the cache and so there is always a cache miss when trying to access one of them. This leads to A and B both having 256 cache misses and 0 cache hits.

Array C is mapped to an area in cache that is not occupied by Array A or B and so is cached effectively. It has 4 cache misses and 252 cache hits.

In total this is 516 cache misses and 252 cache hits.

2.2 fully-associative cache

2.2.1 sizes

Offset : The same as direct mapped cache $= 12b$

Index : Fully associative cache dose not have an index $0b$

Tag : $size(\text{tag}) = addresssize - size(\text{offset}) = 32 - 12 = 20b$

2.2.2 hits and misses

With a fully-associative cache blocks can be placed anywhere, this means we avoid the overwriting problems we saw with the direct mapped cache.

Each array is 256, 32 bit, words. Giving a total size of exactly 8Kb. The cache is much larger than this (16kB) and so can easily fit all three arrays in cache at once. (We only need to be able to store 3 blocks, the current block of each array, to be able to effectively cache the arrays)

This means that there will only ever be cache misses when a new block of an array needs to be loaded. Since each Array takes up 4 blocks, each array will have 4 cache misses and 252 cache hits

In total this is 12 cache misses and 756 cache hits. Which is a big improvement over the direct mapped cache.

2.3 set-associative cache with 2 blocks per set with a a not-last-used replacement policy

2.3.1 sizes

Offset : The same as direct mapped cache $= 12b$

Index : $size(\text{index}) = log_2(\text{numsets}) - 1 = log_2(\text{blocksize}/2) - 1 = 4$

Tag : $size(\text{tag}) = addresssize - size(\text{offset}) - size(\text{index}) = 32 - 12 - 4 = 16b$

Sets : $numSets = numblocks/blocksPerSet = 64/2 = 32sets$

2.3.2 hits and misses

Again I used excel to help automate the process as mush as possible, by generating the cache block location for the array's physical block locations.

Table 2: block placement in a set-associative cache with 2 blocks per set with a not-last-used replacement policy

A	Block Offset				Total	
	0	64	128	192	Miss	Hit
majority hit/miss	m	m	m	m	256	0
phys addr (byte)	2711684608	2711684864	2711685120	2711685376		
phys block (block)	10592518	10592519	10592520	10592521		
cache Set (set)	6	7	8	9		
cache Set block (block)	0/1	0/1	0/1	0		

B	Block Indexes				Total	
	0	64	128	192	Miss	Hit
majority hit/miss	h	h	h	h	256	0
phys addr (byte)	2711750144	2711750400	2711750656	2711750912		
phys block (block)	10592774	10592775	10592776	10592777		
cache Set (set)	6	7	8	9		
cache Set block (block)	0/1	0/1	0/1	1		

C	Block Indexes				Total	
	0	64	128	192	Miss	Hit
majority hit/miss	m	m	m	m	256	0
phys addr (byte)	3165496832	3165497088	3165497344	3165497600		
phys block (block)	12365222	12365223	12365224	12365225		
cache Set (set)	6	7	8	9		
cache Set block (block)	0/1	0/1	0/1	0		

Although surprising at first this cache does not manage to effectively cache any of the arrays. Because all three of the arrays map top the same cache set, they end up over writing each other due to the cache replacement policy. E.g. If a block of A block 0 of the set, then B goes in position 1. When C is placed into the set it is full so the cache replacement policy is used, and so A is replaced with C. Then we try to access A which is not in the array so there is a miss and B gets overwritten with A. And so on ...

In total there are 768 cache misses and 0 cache hits.

2.4 set-associative cache with 8 blocks per set with a least-recently-used replacement

2.4.1 sizes

Offset : The same as direct mapped cache $= 12b$

Index : $size(\text{index}) = log_2(\text{numsets}) - 1 = log_2(\text{blocksize}/8) - 1 = 2$

Tag : $size(\text{tag}) = addresssize - size(\text{offset}) - size(\text{index}) = 32 - 12 - 2 = 18b$

Sets : $numSets = numblocks/blocksPerSet = 64/8 = 8sets$

2.4.2 hits and misses

Table 3: block placement in a set-associative cache with 8 blocks per set with a least-recently-used replacement policy

A	Block Offset				Total	
	0	64	128	192	Misses	Hits
majority hit/miss	m	m	m	m	4	252
phys addr (byte)	2711684608	2711684864	2711685120	2711685376		
phys block (block)	10592518	10592519	10592520	10592521		
cache Set (set)	6	7	0	1		
cache Set block (block)	0	0	0	0		

B	Block Index				Total	
	0	64	128	192	Misses	Hits
majority hit/miss	m	m	m	m	4	252
phys addr (byte)	2711750144	2711750400	2711750656	2711750912		
phys block (block)	10592774	10592775	10592776	10592777		
cache Set (set)	6	7	0	1		
cache Set block (block)	1	1	1	1		

C	Block Index				Total	
	0	64	128	192	Misses	Hits
majority hit/miss	m	m	m	m	4	252
phys addr (byte)	3165496832	3165497088	3165497344	3165497600		
phys block (block)	12365222	12365223	12365224	12365225		
cache Set (set)	6	7	0	1		
cache Set block (block)	2	2	2	2		

Here A, B and C map to the same set, but there are 8 blocks per set so the current block of all three can be kept in memory at the same time with out overwriting each other. This means that all three Arrays can be effectively cached and should only have cache misses when the next block of the array needs loading.

In total there are 12 cache misses and 756 cache hits.

3 Hazards

Multiplication Algorithm - Original

```
1        loadi r2, 16
2        ori r30, r0, 4
3        loadr r3, r30
4        xori r29, r0, 65535
5        loado r4, r30, 3
6        move r7, r0
7        andi r5, r29, 16
8   L3:  andi r6, r4, 1
9        br r6, =0, +2 (L4)
10       add r7, r3, r7
11  L4:  shr r4, r4, 1
12       shl r3, r3, 1
13       dec r5, r5
14       br r5, !=0, -6 (L3)
15       stori r7, 0
```

3.1 RAW Data Hazards Running On Arch D

This is a five stage pipeline so any accesses to data or branches that depend on data, that was assigned less than five instructions ago are data/control hazards that need to be considered.

1. **Data** line 03 attempts to use the value in $r30$ which is assigned in line 02
2. **Data** line 05 attempts to use the value in $r30$ which is assigned in line 02
3. **Data** line 07 attempts to use the value in $r29$ which is assigned in line 04
4. **Data** line 08 attempts to use the value in $r4$ which is assigned in line 05
5. **Control** line 09 attempts to use the value in $r6$ which is assigned in line 08
6. **Control** line 14 attempts to use the value in $r5$ which is assigned in line 03

3.2 Eliminating hazards without forwarding

By moving the instruction that depend only on $r0$ to be the very first instruction we can reduce the time spent in stall substantially. The *ori* that assigns a value to $r30$ is move to the very top as it is the first register that another instruction depends on. next comes the *xori* that assigns a value to $r29$ as it is the next register than another value depends on. The instruction that loads a value into $r2$ is an odd one because the value in $r2$ is

never used within the multiplication algorithm. This means we can essentially use it as a *nop* instruction, placing it before the instruction that depends on the value of $r30$.

Multiplication Algorithm - Eliminated Some Data Hazards Without Forwarding Or Nops

```
1       ori r30, r0, 4
2       xori r29, r0, 65535
3       move r7, r0
4       loadi r2, 16
5       loado r4, r30, 3
6       loadr r3, r30
7       andi r5, r29, 16
8  L3:  andi r6, r4, 1
9       br r6, =0, +2 (L4)
10      add r7, r3, r7
11 L4:  shr r4, r4, 1
12      shl r3, r3, 1
13      dec r5, r5
14      br r5, !=0, -6 (L3)
15      stori r7, 0
```

Adding a *nop* before the two *load* instruction that depend on $r30$ eliminates that data hazard, along with the data hazard with the *addi* the depends on $r29$.

Multiplication Algorithm - Eliminated Some Data Hazards Without Forwarding, With Nops

```
1       ori r30, r0, 4
2       xori r29, r0, 65535
3       move r7, r0
4       loadi r2, 16
5       nop
6       loado r4, r30, 3
7       loadr r3, r30
8       andi r5, r29, 16
9       nop
10      nop
11 L3:  andi r6, r4, 1
12      br r6, =0, +2 (L4)
13      add r7, r3, r7
14 L4:  shr r4, r4, 1
15      shl r3, r3, 1
16      dec r5, r5
17      br r5, !=0, -6 (L3)
18      stori r7, 0
```

3.3 Eliminating data hazards with forwarding

The data hazard on line 05 which attempts to use the value in $r30$ which is assigned in line 02, can be solved by using the $F2$ path. Which connects the Register Write stage to the Execute stage.

The data hazard on line 07 which attempts to use the value in $r29$ which is assigned in line 04, can be solved by using the $F1$ forwarding stage. Which connects the Memory Access stage to the Execute stage.

The data hazard on line 08 which attempts to use the value in $r4$ which is assigned in line 05, can be solved by using the $F2$ path. Which connects the Register Write stage to the Execute stage.

3.4 Branch prediction

It takes 9 clock cycles to get from line 01 up to and including line 09

It will also take 5 clock cycles for line 15's instruction to propagate all the way threw the pipeline.

3.4.1 Without any form of speculative execution

If $B1$ is taken then it takes $4+3=7$ clock cycles to get to $B2$ otherwise it takes $5+3=8$ clock cycles to get to $B2$.

The assessment states that the $B1$ branch is taken 50% which means that 8 times it will take 7 clock cycles and 8 times it will take 8 clock cycles. Giving a total of $8\times 7+8\times 8 = 56+64 = 120$ clock cycles.

If $B2$ is taken then it takes $1+3=4$ clock cycles to get to $B1$ again otherwise it takes 3 clock cycles to get to line 15.

The assessment states that the $B2$ branch is taken $\frac{15}{16}$ times. This gives a total of $15\times 4+1\times 3=64$ clock cycles.

Combining all four of these clock cycle counts gives us the number 0of clock cycles the program will take to execute. This gives a total of $9+120+63+5=197$ clock cycles.

3.4.2 With speculative execution using static prediction – predict not taken

If $B1$ is taken then it takes $4+3=7$ clock cycles (the same as no prediction, because we predicted wrong!), otherwise, if $B1$ is not taken, it takes 5 clock cycles (Less than no prediction because we predicted right!).

This gives a total of $8 \times 7 + 8 \times 5 = 56 + 40 = 96$ clock cycles.

If $B2$ is taken then it takes $1 + 3 = 4$ clock cycles to get to $B1$ again otherwise it takes 1 clock cycles to get to line 15.

This gives a total of $15 \times 4 + 1 \times 1 = 61$ clock cycles.

Combining all four of these clock cycle counts gives us the number 0of clock cycles the program will take to execute. This gives a total of $9+96+61+5 = 171$ clock cycles.

3.4.3 With speculative execution using static prediction – predict taken

If $B1$ is taken then it takes 4 clock cycles (Less than no prediction because we predicted right!), otherwise, if $B1$ is not taken, it takes $5 + 3 = 8$ clock cycles(the same as no prediction, because we predicted wrong!).

This gives a total of $8 \times 4 + 8 \times 8 = 32 + 64 = 96$ clock cycles.

If $B2$ is taken then it takes 1 clock cycles to get to $B1$ again otherwise it takes 3 clock cycles to get to line 15.

This gives a total of $15 \times 1 + 1 \times 3 = 18$ clock cycles.

Combining all four of these clock cycle counts gives us the number 0of clock cycles the program will take to execute. This gives a total of $9+96+18+5 = 128$ clock cycles.

3.4.4 With speculative execution using static prediction – direction-based prediction

Coincidently(which either means I've messed up or you've designed it this way) direction-based speculation gives the same result as predict taken. Because $B1$ branch has the same clock cycle cost in both predict taken and predict not taken modes.

$B1$ has a positive jump so we are using the predict not taken method on it.

If $B1$ is taken then it takes $4 + 3 = 7$ clock cycles (the same as no prediction, because we predicted wrong!), otherwise, if $B1$ is not taken, it takes 5 clock cycles (Less than no prediction because we predicted right!).

This gives a total of $8 \times 7 + 8 \times 5 = 56 + 40 = 96$ clock cycles.

$B2$ has a negative jump so we are using the predict taken method on it.

If $B2$ is taken then it takes 1 clock cycles to get to $B1$ again otherwise it takes 3 clock cycles to get to line 15.

This gives a total of $15 \times 1 + 1 \times 3 = 18$ clock cycles.

Combining all four of these clock cycle counts gives us the number 0of clock cycles the program will take to execute. This gives a total of $9+96+18+5 = 128$ clock cycles.