ECE4750

Evaluating a Dot Product Microbenchmark

Analyzing the Average Memory Access Latency.

TinyRV1 Single-Issue Scalar Processor

TinyRV1 Dual-Issue Superscalar Processor

Two-Way Set Associative Cache

Problem 1. Evaluating a Dot Product Microbenchmark

In this problem, we will explore a dot product microbenchmark executing on a single-issue scalar processor and a dual-issue superscalar processor integrated with either a direct-mapped or set-associative data cache. Here is the C code for the microbenchmark:

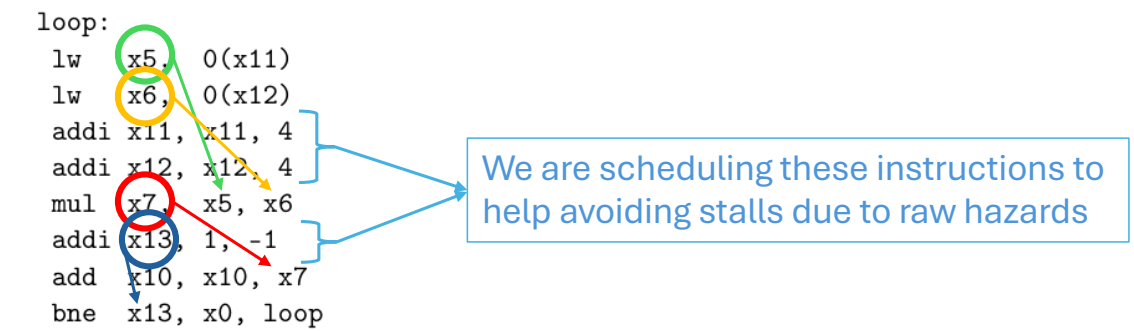
```
int dot( int* a, int* b, int n )
{
  int result = 0;
  for ( int i = 0; i < n; i++ )
    result += a[i] * b[i];
  return result;
}
```

And here is the corresponding assembly:

```
 addi x10, 0, 0

loop:
 lw   x5,  0(x11)
 lw   x6,  0(x12)
 addi x11, x11, 4
 addi x12, x12, 4
 mul  x7,  x5, x6
 addi x13, 1, -1
 add  x10, x10, x7
 bne  x13, x0, loop

 jr   x1
```



Make sure you understand the connection between the C program and assembly before continuing.

For this problem, you should assume a fully bypassed processor that implements the TinyRV1 instruction set. You should assume there an instruction cache with a single-cycle hit latency and a 100% hit rate. You should assume a 256B data cache with 16B cache lines, parallel-read/pipelined-write, a write-back/write-allocate write policy, and a miss penalty of two cycles. Assume the data cache is initially empty.

Assume that we call the `dot` function with two arrays each with 64 elements (i.e., `n` is 64). Assume the base address of array `a` is `0x1000` and the base address of array `b` is `0x2000`.

Instruction cache (I-cache)

- Implements TinyRV1 instruction set.
- Single-cycle hit latency (each instruction fetch takes 1 cycle on a hit).
- **100% hit rate** (no instruction cache misses at all).

- The dot function is called with two arrays, each containing **64 elements** (i.e., n = 64).
- The base address of array a is **0x1000**.
- The base address of array b is **0x2000**.

Data cache (D-cache)

- Total size: **256 B**.
- Cache line (block) size: **16 B.**
- Parallel-read / pipelined-write organization.
- Write policy: **write-back.**
- Allocation policy: write-allocate on a write miss.
- Miss penalty: **2 cycles** per miss.
- Initially empty at the start of the problem.

Assume we are using a direct-mapped cache. **Fill in the following table for data memory accesses corresponding to the load instructions.** Use h or m to indicate a cache hit or miss. Use the set columns to indicate the state of the tag array at the *beginning* of each transaction.

rd/wr	address	tag	idx	h/m	Set 0	Set 1	Set 2	Set 3
rd	0x1000	10	0	m	-	-	-	-
rd	0x2000	20	0	m	10			
rd	0x1004	10	0	m	20			
rd	0x2004	20	0	m	10			
rd	0x1008	10	0	m	20			
rd	0x2008	20	0	m	10			
rd	0x100C	10	0	m	20			
rd	0x200C	20	0	m	10			
rd	0x1010	10	1	m	20			
rd	0x2010	20	1	m		10		
rd	0x1014	10	1	m		20		
rd	0x2014	20	1	m		10		
						20		

Data cache (D-cache)

- Direct-mapped
- Total size: **256 B**.
- Cache line (block) size: **16 B.**
  - Number of lines: 256 B / 16 B = **16 lines**.
- Parallel-read / pipelined-write organization.
- Write policy: **write-back.**
- Allocation policy: write-allocate on a write miss.
- Miss penalty: **2 cycles** per miss.
- Initially empty at the start of the problem.

Now use your table to estimate the average memory access latency for data memory accesses in this microbenchmark.

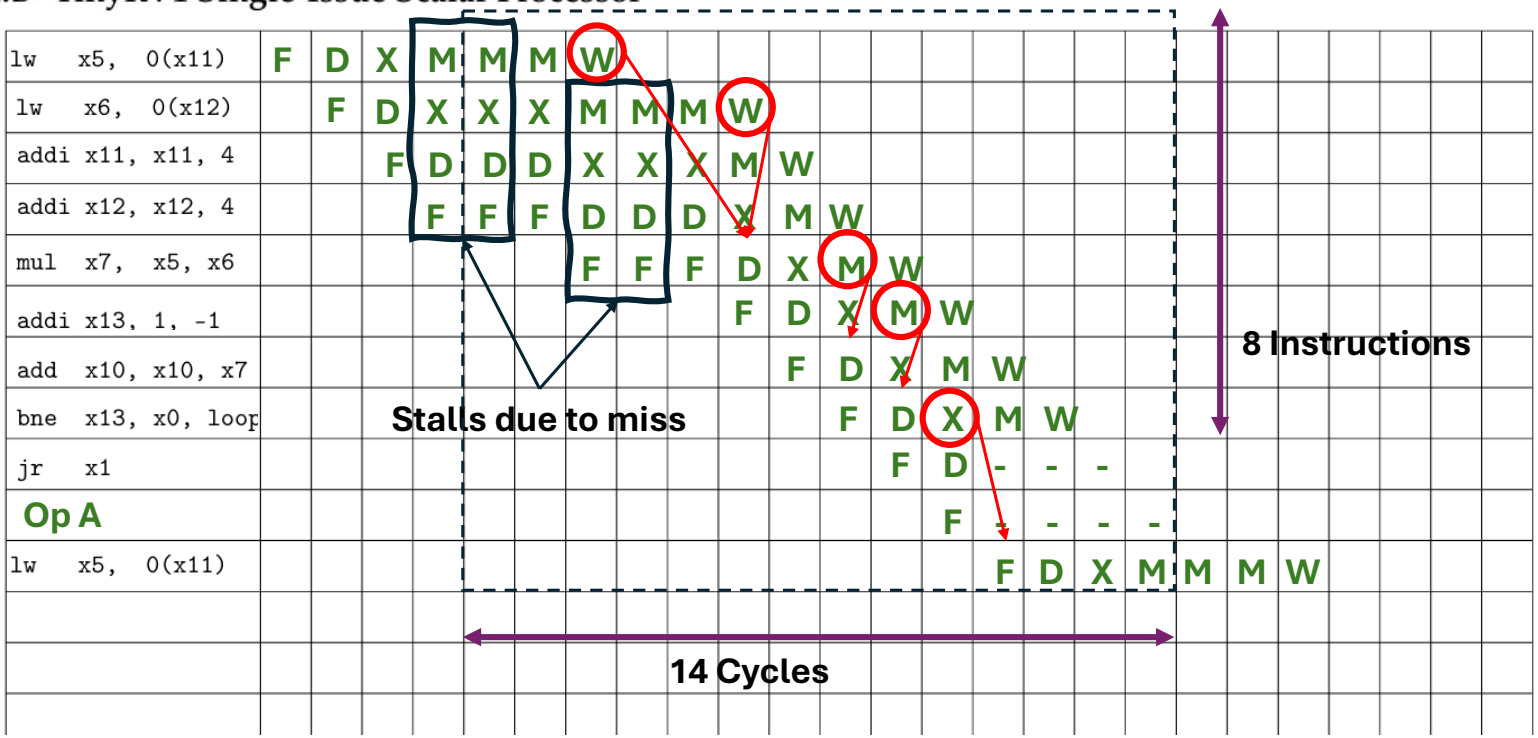
AMAL = Hit Latency + Miss Rate * Miss Penalty

= 1 + 1 * 2 = 3 cycles

Part 1.B TinyRV1 Single-Issue Scalar Processor

Consider the cannonical five-stage fully bypassed TinyRV1 *single-issue scalar* processor integrated with a direct-mapped cache. **Draw a pipeline diagram that illustrates the execution of this loop. Show as many iterations as you need to find the steady state execution.** Only put the instruction name (i.e., `lw`, `addi`, etc) not the full assembly instruction in the pipeline diagram. **Add arrows to your pipeline diagram to indicate all microarchitectural RAW dependencies and any microarchitectural control dependencies (other than those that simply result in fetching the next instruction).**

Part 1.B TinyRV1 Single-Issue Scalar Processor



How long in cycles will it take to execute the vector-vector add example assuming n is 64? What is the CPI?

---

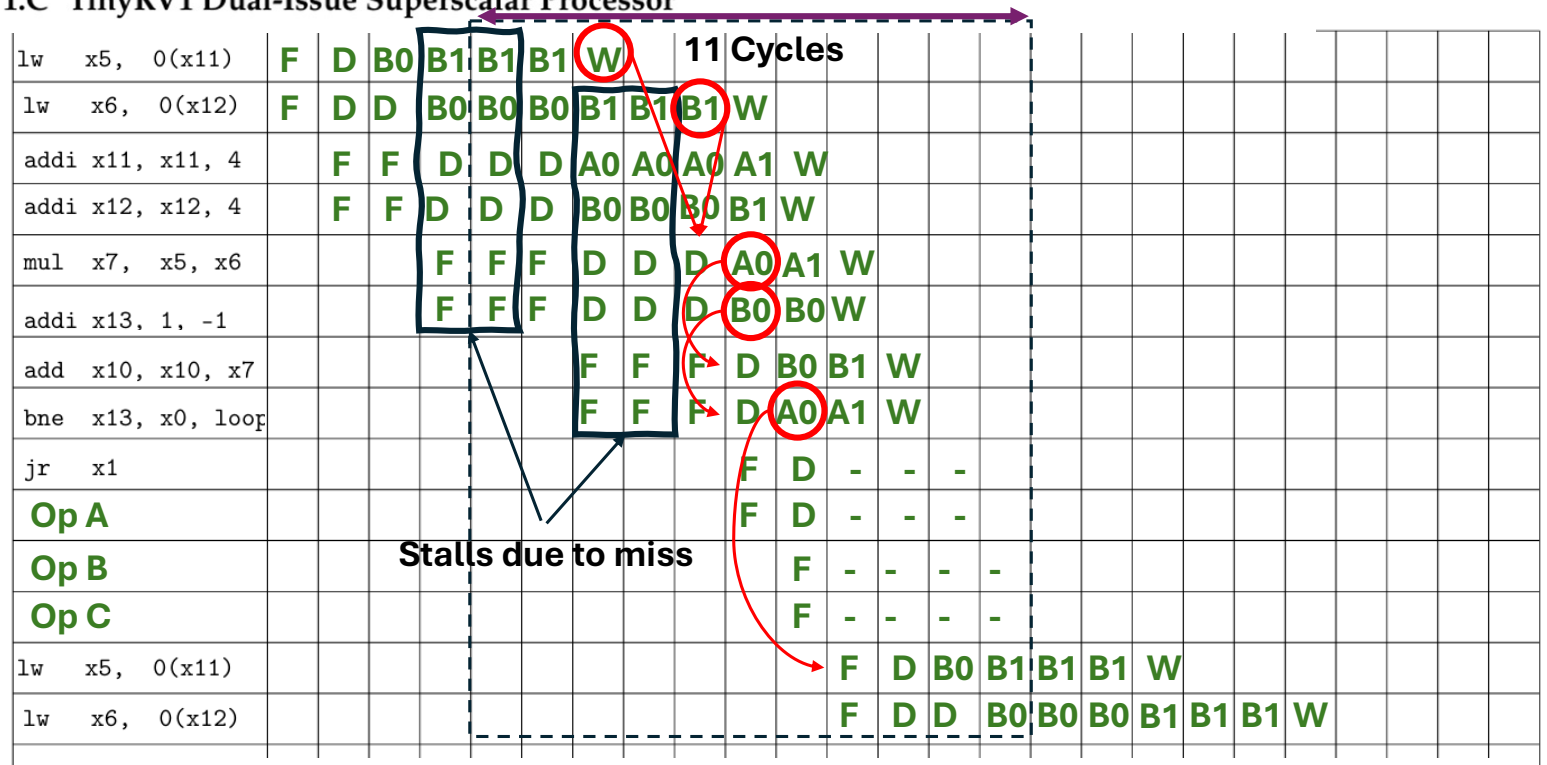
n = 64 => Total cycles =14 Cycles * 64 =896 Cycles

CPI = (14*64)/(8*64)=14/8=1.75

Part 1.C TinyRV1 Dual-Issue Superscalar Processor

Consider the cannonical five-stage fully bypassed TinyRV1 *dual-issue superscalar* processor with indivdual A and B pipes integrated with a direct-mapped cache. Recall that MUL/BNE instructions must use the A pipe, LW/SW instructions must use the B pipe, and ADD/ADDI/JAL/JR instructions can use either pipe. **Draw a pipeline diagram that illustrates the execution of this loop. Show as many iterations as you need to find the steady state execution.** Only put the instruction name (i.e., `lw`, `addi`, etc) not the full assembly instruction in the pipeline diagram. **Add arrows to your pipeline diagram to indicate all microarchitectural RAW dependencies and any microarchitectural control dependencies (other than those that simply result in fetching the next instruction).**

Part 1.C TinyRV1 Dual-Issue Superscalar Processor



How long in cycles will it take to execute the vector-vector add example assuming n is 64? What is the CPI? What is the speedup compared to a single-issue processor?

n = 64 => Total cycles =11 Cycles * 64 =704 Cycles

CPI = (11*64)/(8*64)=11/8=1.375

Speedup = (14*64)/(11*64)=14/11=1.27

Part 1.D Two-Way Set Associative Cache

Start by filling in the following table with your results so far. Then consider replacing the direct-mapped data cache with a two-way set-associative cache. **Use your results from the previous parts to quickly estimate the new CPI when using a set-associative cache and fill those results into this table. Justify your answers.** Discuss some of the trade-offs between these four different configurations.

Processor µArch	Cache µArch	CPI
Single-Issue	Direct-Mapped	**1.75**
Single-Issue	Two-Way Set Assoc	**1.375**
Dual-Issue Superscalar	Direct-Mapped	**1.375**
Dual-Issue Superscalar	Two-Way Set Assoc	**1**

Single Issue:

In iteration 0 => 14 cycles (still compulsory misses)

In Iteration 1,2,3 => 10 cycles (since there is not any stall due to misses)

=> Average cycles per iteration = 14*0.25 + 10*0.75=11 cycles/iter. => CPI= 11/8=1.375

Dual Issue

Iteration 0 => 11 cycles (still compulsory misses)

Iteration 1,2,3 => 7 cycles (since there is not any stall due to misses)

=> Average cycles per iteration = 11*0.25 + 7*0.75=8 cycles/iter. => CPI= 8/8=1