Project 5: Optimizer

Jason Ansel

Overview

- Project guidelines
- Benchmarking Library
- OoO CPUs

Project Guidelines

- Use optimizations from lectures as your arsenal
  - If you decide to implement one, look at Whale / Dragon
- Use the provided programs to substantiate your implementation decisions.
  - Benchmark the provided programs on the target architecture.
  - Hand-implement the transformation first.
  - Don't waste time with ineffectual transformations.
- Cover all of the transformations discussed in class, at the very least qualitatively.

Project Guidelines

- An analysis of each optimization considered `(you can group optimizations if you feel they are symbiotic).
- Give reasons for your benchmarking results given your knowledge of the target architecture.
- Analyze your generated assembly. Look for non-traditional peephole optimizations.
- For final report, describe your full optimizations option and the ideas/experiments behind the implementation.
  - Phase order, convergence?

Project Guidelines

- Use GCC for experimentation
  - In many cases **you** can do better
- Writeup needs to be very detailed
- For each optimization:
  1. **Prove** that it is a win by hand applying and benchmarking
  2. Give a detailed explanation of the implementation
  3. Argue that it is general and correct
- Just Step 1 for Design Proposal

Library

- You are provided with a library
  - lib6035.a
  - In /mit/6.035/provided/lib
- You need to link against it and pthreads
  - gcc4 program.s lib6035.a –lpthread
  - Important to put your .s first!
- Gives you ability to accurately benchmark your code, spawn and join multiple threads and read/write images (pgm)

Benchmarking

- We will use wall time to run your benchmark
  - Measured in us
- You can benchmark the entire program and also a section of code you denote
- Use `start_caliper` and `end_caliper` calls
  - The library will print out the time passed between the calls

Benchmarking Example

```
class Program {
  ...
  void main () {
    read ();
    callout("start_caliper");
    invert ();
    callout(“end_caliper");
    write ();
  }
}
```

```
silver % ./negative
Timer: 1352 usecs
silver %
```

AMD Opteron 270

- ~233 million transistors
- 90nm
- 2.0 GHz
- Dual Core
- 2 Processors per board
- 64 KB L1 Ins Cache
- 64 KB L1 data cache
- 2 MB on-chip L2 cache
- 12 Integer pipeline stages

Opteron's Key Features

- Multiple issue (3 instructions per cycle)
- Register renaming
- Out-of-order execution
- Branch prediction
- Speculative execution
- Load/Store buffering
- 2x Dual core

What is a Superscalar?

- Any (scalar) processor that can “execute” more that one instruction per cycle.
- Multiple instructions can be fetched per cycle
- Decode logic can decide which instructions are independent
  - Decide when an instruction is ready to fire
- Multiple functional units
- All this for instruction-level parallelism!

Multiple Issue

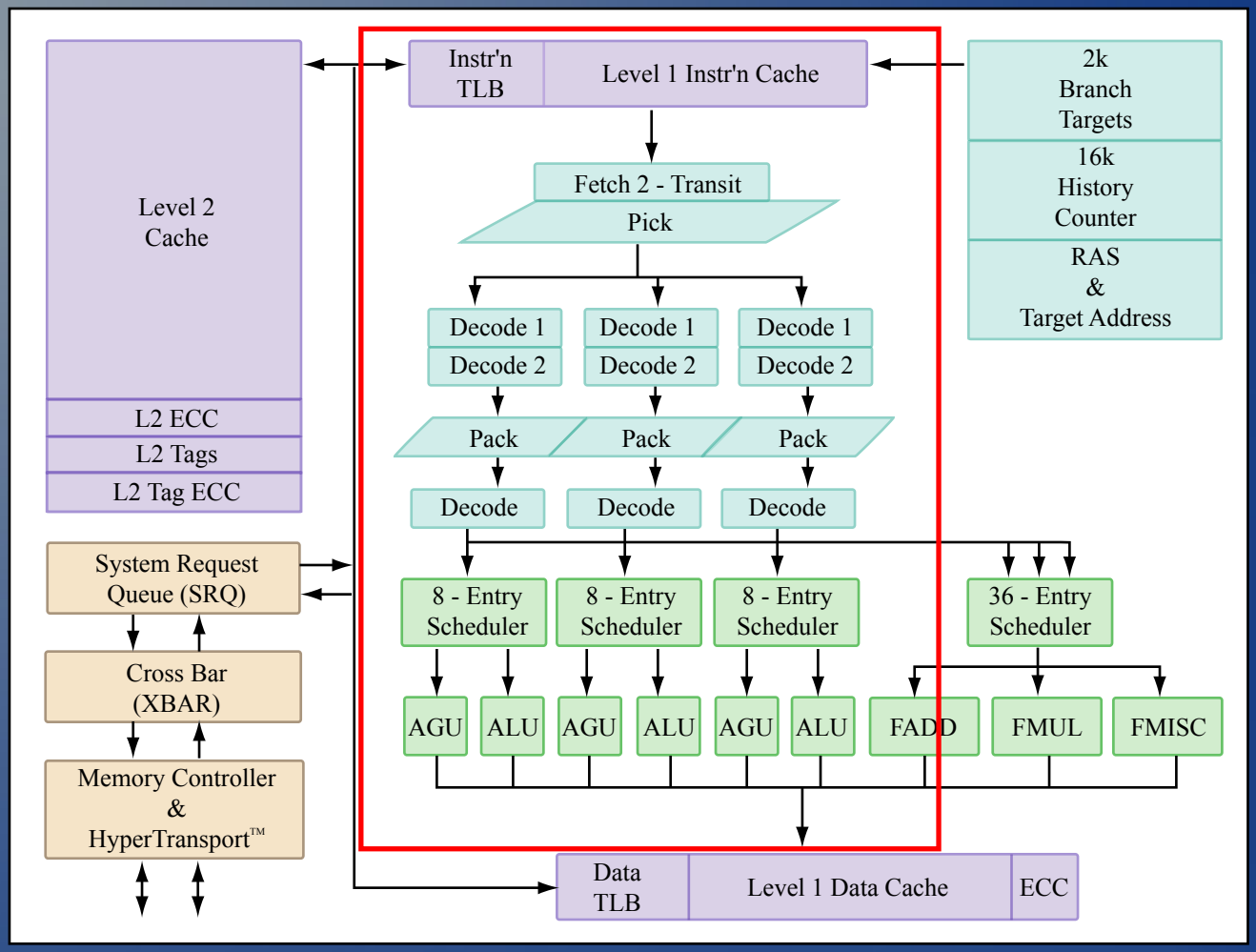


Image by MIT OpenCourseWare.

- The Opteron is a 3-way superscalar:
  - decode & execute & retire 3 **x86-64 instructions** per cycle

Dependencies

- Remember there are 3 types of dependencies:
  - True dependence
    - read after write (RAW)
  - Anti-dependence
    - write after read (WAR)
  - Output dependence
    - write after write (WAW)

```
a = b + c
c = a + b
...
c = g + h
```

Register Renaming

- Used to eliminate artificial dependencies
  - Anti-dependence (WAR)
  - Output dependence (WAW)
- Basic rule
  - All instructions that are in-flight write to a distinct destination register

Register Renaming

- Registers of x86-64 ISA (%rax, %rsi, %r11, etc) are logical registers
- When an instruction is decoded, its logical register destination is assigned to a physical register
- The total number of physical (rename) registers is:
  - instructions_inflight + architectural regs =
  - 72 + 16 = 88 (plus some others)

Register Renaming

- Hardware maintains a mapping between the logical regs and the rename regs
  - Reorder Buffer (72 entries)
- Architectural state is in 16 entry register file
  - 2 entries for each logical register
  - One speculative and one committed
  - Maintains architectural visible state
  - Used for exceptions and miss-prediction

Out-of-Order Execution

- The Opteron can also execute instructions out of program order
  - Respect the data-flow (RAW) dependencies between instructions
- Hardware schedulers executes an instruction whenever all of its operands are available
  - Not program order, dataflow order!
- The Reorder Buffer orders instructions back into program order

Out-of-Order Execution

- When an instruction is placed in the reorder buffer:
  - Get its operands
    - if no inflight writer of operand, get it from architectural register
    - if inflight writer, get writer's ID from state and wait for value to be produced
  - Rename dest register, update state to remember this is most recent writer of dest
- Reorder Buffer:
  - Forward results of instruction to consumers in reorder buffer
  - Write value to speculative architectural register

Branch Prediction

- The outcome of a branch is known late in the pipeline
- We don’t need to stall the pipeline while the branch is being resolved
- Use local and global information to predict the next PC for each instruction
  - Return address stack for ret instruction
- Extremely accurate for integer codes
  - >95% in studies
- Pipeline continues with the predicted PC as its next PC

Speculative Execution

- Continue execution of program with predicted next PC for conditional branches
- An instruction can commit its result to architectural registers
  - In order: only after every prior instruction has committed
  - Means that slow instruction (ex. cache miss) will make everybody else wait!
- 3 instructions can be retired per cycle

Speculative Execution

- Each instruction that is about to commit is checked:
  - Make sure that no exceptions have occurred
  - Make sure that no branch miss-predictions have occurred (for conditional branches)
- If a miss-prediction or an exception occurs
  - Flush the pipeline
  - Clear the reorder buffer and schedulers
  - Forget any speculative architectural registers
  - Forward the correct PC to the fetch unit for miss-prediction
  - Call exception handler for exception

Main Memory

- It is very expensive to go to main memory!
- Optimize for the memory hierarchy
- Keep needed values as close to the processor as possible
- Your control:
  - 16 Logical Regs
  - Main Memory
- Under the hood
  - Rename Regs -> Logical Regs -> L1 -> L2 -> Memory

Load/Store Buffering

- 3 cycle latency for loads that hit in L1 d-cache
- Load/Store Buffer with 12 entries
  - Handles memory requests
  - Entry for each load/store
    - They wait until their address is calculated
  - Loads check the L/S buffer before accessing the
    - If address is in the L/S buffer, get the value from that entry
    - Otherwise, the oldest load probes the cache
  - Stores just wait for their value to be ready
    - Cannot write to cache because it might be speculative
      - Stores must be retired to write to memory
- Complex mechanisms for maintaining program order of loads and stores

Four Cores!

- Dual cores per processor, dual processor per board
- You can map different iterations of a forpar loop to different cores
  - Data-level parallelism
- Memory coherence is maintained by a global cache-coherence mechanism
  - Snooping mechanism
  - All cores see same state of d-cache
- More on this in the future!

Example

```
//load array's last addr into %rdi
loop:
  mov (%rdi), %r10
  add %r11, %r10
  mov %r10, (%rdi)
  sub $8, %rdi
  bge loop
```

Example

Load/Store Buffer

ID	Op	Addr	Value
1			
2			
3			
4			
5			
6			

Architectural State

Reg	Committed	Speculative
%rdi	80	
%r10		
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1				
2				
3				
4				
5				
6				
7				
8				

ID	Op	Dest	Src1	Src2
9				
10				
11				
12				
13				
14				
15				
16				

Example

Load/Store Buffer

Architectural State

Reg	Committed	Speculative
%rdi	80	
%r10		
%r11	5	

ID	Op	Addr	Value
1			
2			
3			
4			
5			
6			

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		
2	add	r10	RB1	r11
3	mov		RB2	
4	sub	rdi	rdi	$8
5	bge	loop	RB4	
6				
7				
8				

ID	Op	Dest	Src1	Src2
9				
10				
11				
12				
13				
14				
15				
16				

Example

Load/Store Buffer

ID	Op	Addr	Value
1	ld	80	??
2	st	80	RB2
3			
4			
5			
6			

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Architectural State

Reg	Committed	Speculative
%rdi	80	
%r10		
%r11	5	

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	sub	rdi	rdi	$8
5	bge	loop	RB4	
6				
7				
8				

ID	Op	Dest	Src1	Src2
9				
10				
11				
12				
13				
14				
15				
16				

Example

Load/Store Buffer

ID	Op	Addr	Value
1	ld	80	??
2	st	80	RB2
3			
4			
5			
6			

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Architectural State

Reg	Committed	Speculative
%rdi	80	
%r10		
%r11	5	

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	sub	rdi	rdi	$8
5	bge	loop	RB4	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	RB4	$8
10	bge	loop	RB9	
11				
12				
13				
14				
15				
16				

Example: "Cycle 1"

Architectural State

Reg	Committed	Speculative
%rdi	80	
%r10		
%r11	5	

Load/Store Buffer

ID	Op	Addr	Value
1	ld	80	??
2	st	80	RB2
3	ld	RB4	??
4	st	RB4	RB7
5	ld	RB9	??
6	st	RB9	RB12

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	sub	rdi	rdi	$8
5	bge	loop	RB4	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	RB4	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 2"

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3	ld	RB4	??
4	st	RB4	RB7
5	ld	RB9	??
6	st	RB9	RB12

Architectural State

Reg	Committed	Speculative
%rdi	80	72 (RB4)
%r10		
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	72			
5	bge	loop	RB4	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	RB4	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 2"

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3	ld	72	??
4	st	72	RB7
5	ld	RB9	??
6	st	RB9	RB12

Architectural State

Reg	Committed	Speculative
%rdi	80	72 (RB4)
%r10		
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	72			
5	bge	loop	72	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	72	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 2"

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3	ld	72	??
4	st	72	RB7
5	ld	RB9	??
6	st	RB9	RB12

Architectural State

Reg	Committed	Speculative
%rdi	80	72 (RB4)
%r10		
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		LS1
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	72			
5	bge	loop	72	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	72	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 2"

Architectural State

Reg	Committed	Speculative
%rdi	80	72 (RB4)
%r10		
%r11	5	

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3	ld	72	??
4	st	72	RB7
5	ld	RB9	??
6	st	RB9	RB12

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		10
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	72			
5	bge	loop	72	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	72	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 2"

Architectural State

Reg	Committed	Speculative
%rdi	80	72 (RB4)
%r10		
%r11	5	

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3	ld	72	??
4	st	72	RB7
5	ld	RB9	??
6	st	RB9	RB12

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	mov	r10		10
2	add	r10	RB1	r11
3	mov	LS2	RB2	
4	72			
5	bge	loop	72	
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	sub	rdi	72	$8
10	bge	loop	RB9	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	RB9	$8
15	bge	loop	RB14	
16				

Example: "Cycle 3"

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	RB2
3			9
4	st	72	RB7
5	ld	64	??
6	st	RB9	RB12

Architectural State

Reg	Committed	Speculative
%rdi	80	64 (RB9)
%r10	10 (RB1)	
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	10			
2	add	r10	10	r11
3	mov	LS2	RB2	
4	72			
5				
6	mov	r10		LS3
7	add	r10	RB6	r11
8	mov	LS4	RB7	

ID	Op	Dest	Src1	Src2
9	64			
10	bge	loop	64	
11	mov	r10		LS5
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	64	$8
15	bge	loop	RB14	
16				

Example: "Cycle 4"

Load/Store Buffer

ID	Op	Addr	Value
1			10
2	st	80	15
3			9
4	st	72	RB7
5			8
6	st	RB9	RB12

Architectural State

Reg	Committed	Speculative
%rdi	80	64 (RB9)
%r10	15 (RB2)	
%r11	5	

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	10			
2	15			
3	mov	LS2	15	rdi
4	72			
5				
6	9			
7	add	r10	9	r11
8	mov	LS4	RB7	

ID	Op/Val	Dest	Src1	Src2
9	64			
10	bge	loop	64	
11	mov	r10		8
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	sub	rdi	64	$8
15	bge	loop	RB14	
16	mov	r10	RB14	LS7

Example: "Cycle 5"

Architectural State

Reg	Committed	Speculative
%rdi	80	64 (RB9)
%r10	15 (RB2)	
%r11	5	

Load/Store Buffer

ID	Op	Addr	Value
1			10
2			
3			9
4	st	72	14
5			8
6	st	RB9	RB12

```
loop:
 mov (%rdi), %r10
 add %r11, %r10
 mov %r10, (%rdi)
 sub $8, %rdi
 bge loop
```

Reorder Buffer

ID	Op/Val	Dest	Src1	Src2
1	10			
2	15			
3				
4	72			
5				
6	9			
7	14			
8	mov	LS4	14	

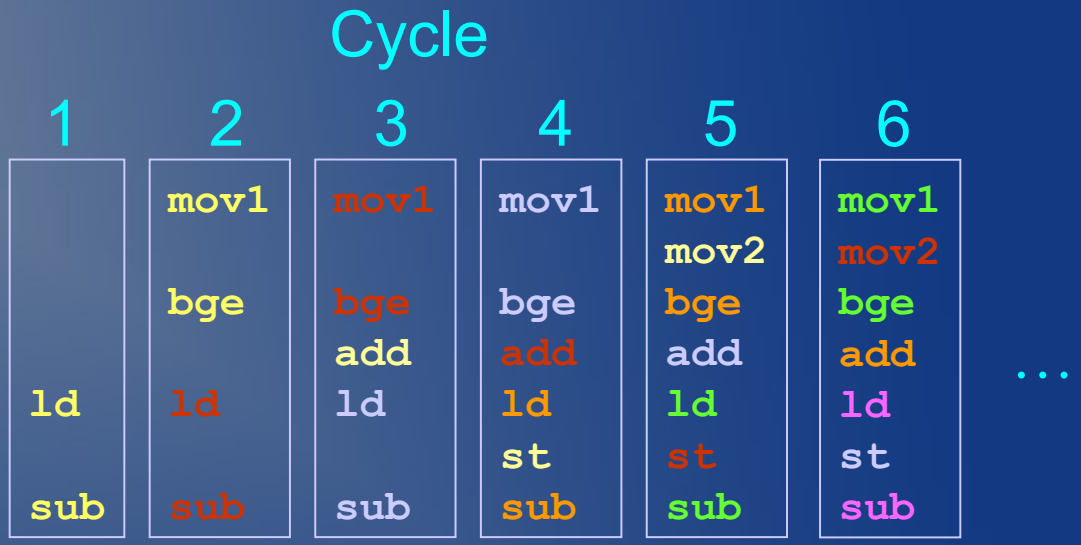
ID	Op/Val	Dest	Src1	Src2
9	64			
10				
11	mov	r10		8
12	add	r10	RB11	r11
13	mov	LS6	RB12	
14	56			
15	bge	loop	56	
16	mov	r10	56	LS7

Example: Steady State

Instructions Decoded:

```
loop: //iteration 1
 ld
 mov1
 add
 st
 mov2
 sub
 bge

loop: //iteration 2
…
loop: //iteration 3
…
loop: //iteration 4
…
loop: //iteration 5
…
loop: //iteration 6
…
```



Instructions from 5 iterations can fire in one cycle

Steady-State

- In each cycle:
  - Instructions issued from different iterations of the original loop
- Remember that the Opteron can fire 3 instructions per cycle
  - not 7 like on previous slide
- What does this say about instruction scheduling techniques?
  - List scheduling, trace scheduling, loop unrolling, software pipelining, etc.

Conclusions

- Opteron is doing many things under the hood
  - Multiple issue
  - Reordering
  - Speculation
  - Multiple levels of caching
- Your optimizations should compliment what is going on
- Use your “global” knowledge of the program
  - Redundancy removal
  - Register allocation
  - What else?

6.035 Computer Language Engineering

Spring 2010