ECE 4750 PSET 1

Tim Yao (ty252)

Sept 24, 2015

1 Comparing CISC, RISC, and Stack ISAs

1.a x86 CISC ISA

The x86 program contains 3+2+3+1+2+2=13 bytes.
If the size = 10, the number of bytes of instructions that need to be fetched is 3+2+2+2+10*(3+1+2+2)=89 bytes. The xor and jmp instructions are both executed once at the beginning of the program. The instructions cmp and jl are also executed at the beginning of the program, before the actual looping starts. Every loop contains the two inc instructions, cmp, and jl.
Assuming 32-bit data values, a total of 4*10 = 40 bytes need to be both loaded and stored to the data memory.

1.b MIPS32 RISC ISA

```
1           sll r5, r5, 2      // Multiply the size by 4 due to byte addressed memory
2           jmp L1             // Jump to L1 to do the initial size != 0 check.
3           addi r12, r4, r5   // Calculate the absolute end location of the array in memory
4   loop:   addi r13, r13, 1   // Increment r13 by 1
5           sw r13, 0(r14)     // Store r13 back into the current spot in array
6           addi r4, r4, 4     // Increment the counter r14 by 4 (memory is byte addressed)
7   L1:     bne r12, r4, loop  // If r12 != r4, then go to "loop" and continue through the array
8           lw r13, 0(r4)      // Load the current item in the array into r13
```

There are two types of optimization in the code. The first takes advantage of the branch delay slots after the jump and branch instructions. The second restructures the looping mechanism to reduce the number of calculations and register usage. The addi instruction on line 3 takes advantage of the branch delay slot after the jump instruction on line 2. This is possible because the addi instruction needs to be executed regardless of the jump. The load word instruction on line 8 also takes advantage of the branch delay slot right after the branch instruction on line 7. The load word instruction is used for retrieving one element from the array and place it into the temporary register r13. Normally, this instruction is placed at the beginning of the loop, but because r13 is a temporary register, we can also load into r13 without any consequences (regardless whether the branch is taken or not). During normal looping (when bne is branching), the load word instruction saves a cycle each. When the loop is completed, that instruction will load an element beyond the scope of the array into r13. In this specific situation, this out of bounds load should not cause any problems, because r13 is a temporary register and we will not be using it after the loop has completed.
My MIPS32 program contains 4*8 = 32 bytes (8 instructions with instruction taking up 4 bytes).
A total of 4*(3+2+10*5) = 220 bytes of instructions need to be fetched if size is 10.
Assuming 32-bit data values, 4*10 + 4 = 44 bytes of data need to be loaded from memory. 4*10 = 40 bytes of data need to be stored to memory.

1.c Simple Stack-Based ISA

Contents of Stack on First Iteration	Access Stack Memory?	Label	Instruction
aptr; size			goto L1
aptr; size		loop:	swap
size; aptr	Y		dup
size; aptr; aptr	Y		dup
size; aptr; aptr; aptr			pushm
size; aptr; aptr; mem	Y		pushi(1)
size; aptr; aptr; mem; 1	Y		add
size; aptr; aptr; (mem+1)	Y		popm
size; aptr	Y		pushi(4)
size; aptr; 4	Y		add
size; (aptr+4)			swap
(aptr+4); size;	Y		pushi(-1)
(aptr+4); size; -1	Y		add
aptr; size	Y	L1:	dup
aptr; size; size	Y	loop:	bgtz loop

Figure 1: array_increment loop in stack based ISA
The final stack is (aptr+4); (size-1) after add on 3rd to last row

The stack based ISA program contains 5+1+5+1+1+1+1+2+1+1+2+1+1+2+1=26 bytes.
If size is 10, a total of 5+1+5+10*(1+1+1+1+2+1+1+2+1+1+2+1+1+5)=221 bytes of instructions need to be fetched.
Assuming 32-bit data values, a total of 4*10=40 bytes of data need to be both fetched from and stored to data memory.
11 instructions will result in some kind of access to the special stack memory. If size is 10, the special stack memory needs to be accessed a total of 2+10*11=112 times. If the microarchitecture implemented eight 32-bit registers for the stack, this program will never need to access the special stack memory. This is because the largest the stack will be during execution is 5 elements.

1.d Comparison of ISAs

The x86 ISA should give the smallest static code size and fewest dynamically fetched instruction bytes compared to the MIPS32 and stack based ISA. This is mainly due to it being a CISC and having a variable instruction length. In comparison to the MIPS32, which has all 4-byte instructions, the x86 ISA can reduce both the static and dynamic code size with instructions that are generally less than 4 bytes. In comparison to both the MIPS32 and the stack based ISA, x86 also uses a complex instruction set (CISC), which purpose is to allow a program to be written with the fewest lines of assembly as possible. While this increases the complexity of the microarchitecture and makes the ISA harder to work with, it does allow reduce the number of instructions is needed for a program. As already shown by the array_increment program, x86 uses the fewest instructions. While the stack based ISA also uses variable length instructions, the stack design is not well suited for most programs. The ability to only access the top of the stack requires extensive use of the swap instruction, which decreases the benefits of the variable length instructions.

2 Microcoded PARCv1 Processor

2.a Implementing Conditional Move

		Bus Enables					Register Enables						Mux		Func		RF		mreq		
State	**Pseudo Control Sigs**	**pc**	**iau**	**alu**	**rf**	**rd**	**pc**	**ir**	**a**	**b**	**c**	**wd**	**b**	**c**	**iau**	**alu**	**sel**	**wen**	**val**	**op**	**next**
M0:	A <- RF[rt]	0	0	0	1	0	0	0	1	0	0	0	x	x	x	x	rt	0	0	x	n
M1:	B <- RF[r0]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	r0	0	0	x	n
M2:	A<-RF[rs];goto F0 if A=B	0	0	1	1	0	0	0	1	0	0	0	x	x	x	cmp	rs	0	0	x	b
M3:	RF[rd] <- A	0	0	1	1	0	0	0	0	0	0	0	x	x	x	cpa	rd	1	0	x	f

Figure 2: Microcode table for movn

M0 and M1 puts the values of rt and r0 into registers A and B respectively for comparison by the ALU. This is to check and see if rt is 0. M2 uses the cmp function of the ALU to check if rt is 0 and goto F0 (skip to next instruction) if rt is 0 by setting the next state to b. If rt is not 0, then the FSM will continue to execute states M3 and M4. M3 gets the value of rs and temporarily puts it in register A. M4 moves the value in register A to rd by utilizing the cpa (copy A to output) function of the ALU. This completes the microcode program for movn.

2.b Implementing Memory-Memory Increment Instruction

		Bus Enables					Register Enables						Mux		Func		RF		mreq		
State	**Pseudo Control Sigs**	**pc**	**iau**	**alu**	**rf**	**rd**	**pc**	**ir**	**a**	**b**	**c**	**wd**	**b**	**c**	**iau**	**alu**	**sel**	**wen**	**val**	**op**	**next**
I0:	C <- si(imm)	0	0	0	0	0	0	1	0	0	1	0	x	b	si	x	x	0	0	x	n
I1:	B <- RF[rs]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	rs	0	0	x	n
I2:	A <- RF[r0]	0	0	0	1	0	0	0	1	0	0	0	x	x	x	x	r0	0	0	x	n
I3:	C[0]? A+B:copy A	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
I4:	C[0]? A+B:copy A	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
I5:	C[0]? A+B:copy A	0	0	1	0	0	0	0	1	1	1	0	s	s	x	+?	x	0	0	x	n
I6:	C[0]? A+B:copy A	0	0	1	0	0	0	0	1	0	0	0	x	x	x	+?	x	0	0	x	n
I7:	B <- RF[rt]	0	0	0	1	0	0	0	0	1	0	0	b	x	x	x	rt	0	0	x	n
I8:	mreq_addr<-B<-A+B	0	0	1	0	0	0	0	0	1	0	0	b	x	x	+	x	0	1	r	n
I9:	A <- RD	0	0	0	0	1	0	0	1	0	0	0	x	x	x	x	x	0	0	x	n
I10:	WD <- A+1	0	0	1	0	0	0	0	0	0	0	1	x	x	x	+1	x	0	0	x	n
I11:	mreq_addr <- B	0	0	1	0	0	0	0	0	0	0	0	x	x	x	cpb	x	0	1	w	f

Figure 3: Microcode table for inc

The microinstruction I0 to I2 sets up registers A, B, and C for doing the R[rs] x imm[3:0] multiplication. I3 to I7 uses the +? function of the ALU to perform the multiplication. I7 gets the rt register to prepare for the addition. I8 performs the addition to calculate the memory address. It send that address to both mreq_addr and register B for storage. I9 gets the value returned from the memory and puts it into register A. I10 performs the increment by 1 function and places the result into the WD register. I11 sends the memory address stored in B to the mreq_req address so that the result in WD can be stored back into memory.

3 Multiplier Microarchitecture

	Microarchitecture	**Num Trans** (#)	**Cycle Time** (τ)	**Transaction Latency** (cyc)	(τ)	**Transaction Throughput** (trans/cyc)	(cyc/trans)	**Total Execution Time** (τ)
(a)	1-Cycle	60	62	1	62	1	1	3720
(b)	Iterative	60	20	4	60	0.25	4	3600
(c)	2-Cycle Unpiplined	60	32	2	64	0.5	2	3840
(c)	2-Cycle Pipelined	60	32	2	64	0.98	1.02	1952
(d)	4-Cycle Unpiplined	60	17	4	68	0.25	4	4080
(d)	4-Cycle Piplined	60	17	4	68	0.95	1.05	1071
(e)	Var-Lat Pipelined	60	20	1-5	20-100	0.59	1.7	2040

Figure 4: Evaluation of Various Multiplier Microarchitectures

3.a Iterative Microarchitecture

	Transaction	Cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	`mul 0xdeadbeef, 0xf`	Z	Z	Z	Z												
2	`mul 0xf5fe4fbc, 0x7`					Z	Z	Z	Z								
3	`mul 0x0a01b044, 0x3`									Z	Z	Z	Z				
4	`mul 0xdeadbeef, 0xf`													Z	Z	Z	Z

Figure 5: Iterative Multiplier Transaction vs Time Diagram

3.b Two-Cycle Microarchitecture

	Transaction	Cycle 1	2	3	4	5	6	7	8
1	`mul 0xdeadbeef, 0xf`	X0	X1						
2	`mul 0xf5fe4fbc, 0x7`			X0	X1				
3	`mul 0x0a01b044, 0x3`					X0	X1		
4	`mul 0xdeadbeef, 0xf`							X0	X1

Figure 6: 2 Cycle Unpipelined Transaction vs Time Diagram

	Transaction	Cycle 1	2	3	4	5
1	`mul 0xdeadbeef, 0xf`	X0	X1			
2	`mul 0xf5fe4fbc, 0x7`		X0	X1		
3	`mul 0x0a01b044, 0x3`			X0	X1	
4	`mul 0xdeadbeef, 0xf`				X0	X1

Figure 7: 2 Cycle Pipelined Transaction vs Time Diagram

3.c Two-Cycle Microarchitecture

		Cycle															
	Transaction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	`mul 0xdeadbeef, 0xf`	X0	X1	X2	X3												
2	`mul 0xf5fe4fbc, 0x7`					X0	X1	X2	X3								
3	`mul 0x0a01b044, 0x3`									X0	X1	X2	X3				
4	`mul 0xdeadbeef, 0xf`													X0	X1	X2	X3

Figure 8: 4 Cycle Unpipelined Transaction vs Time Diagram

		Cycle						
	Transaction	1	2	3	4	5	6	7
1	`mul 0xdeadbeef, 0xf`	X0	X1	X2	X3			
2	`mul 0xf5fe4fbc, 0x7`		X0	X1	X2	X3		
3	`mul 0x0a01b044, 0x3`			X0	X1	X2	X3	
4	`mul 0xdeadbeef, 0xf`				X0	X1	X2	X3

Figure 9: 4 Cycle Pipelined Transaction vs Time Diagram

3.d Variable-Latency Microarchitecture

		Cycle									
	Transaction	1	2	3	4	5	6	7	8	9	10
1	`mul 0xdeadbeef, 0xf`	Y	X0	X1	X2	X3					
2	`mul 0xf5fe4fbc, 0x7`		Y	Y	X1	X2	X3				
3	`mul 0x0a01b044, 0x3`				Y	Y	X2	X3			
4	`mul 0xdeadbeef, 0xf`						Y	X0	X1	X2	X3

Figure 10: Variable-Latency Microarchitecture Transaction vs Time Diagram

3.e Comparison of Microarchitectures

The 4 cycle pipelined architecture has the highest performance. It provided the best transaction throughput and therefore lowest total execution time. The 4 cycle pipelined multiplier allowed us to execute 4 multiply instructions at the same time provided that there are no data dependencies between them. For this specific type of mlutiplier (32 bit operand by 4 bit operand), anything more than a 4 cycle pipeline will not provide any benefits. The best performance is achieved when the number of cycles in the pipeline is equal to the number of bits in the second operand (ie. a 32 bit by 16 bit multiplier will have a 16-cycle pipeline). The variable latency pipelined multiplier can provide the same transaction throughput as the 4-cycle pipeline and better transaction latency, but only for certain data sets and instruction ordering (ie. a continuous stream of data that only needs the lowest 2 bits). Due to the necessity to stall in the Y stage, the variable latency multiplier resulted in a worse throughput and latency when compared to the 4-cycle pipeline in our tests. With a larger second operand (of n bits) and a more random data set, the variable latency multiplier will have roughly the same transaction throughput as the n-cycle pipeline, but a slightly better transaction latency.

In terms of area, the fixed latency multiplier will have the smallest area while the variable latency pipelined multiplier will have the largest area. The 4-cycle pipelined multiplier, with the second largest area, provided the best performance in this test due to the uniqueness of this data set. I believe that the area difference between these different architectures is not significant because they only involve the addition of registers and muxes, while the amount of combinational logic for arithmetic purposes remained roughly the same (same number of 32-bit adders and shifters).

While the 4 cycle pipelined multiplier provided the best throughput, it also had one of the worst latency. Therefore, if there are data dependencies between a succession of transactions, a fixed latency or variable latency pipelined multiplier will have provided better performance due to their lower transaction latency. In the case of the variable latency multiplier, the performance will vary depending on the data set as stated before.

Overall, for a 32-bit by 4-bit multiplier, the 4 cycle pipelined multiplier will provide the best performance in general. With good software scheduling, we can minimize the penalty for data dependencies and therefore maintain a max throughput. However, if we wish to implement a larger multiplier such as a 32-bit by 16-bit multiplier, we should choose the variable-latency pipelined multiplier instead. With the larger operand, there are more upper bits that we can potentially skip, therefore the performance benefits of skipping them are also greater, especially when there are data dependencies. In that case, it is likely to outperform a 16-cycle pipelined multiplier.

4 Two-Cycle Pipelined Integer ALU and Multiplier

4.a Part 4.A Control and Data Hazard Latencies

Jump Resolution Latency = 2
Branch Resolution Latency = 4
ALU-use Delay Latency = 2
Load-use Delay Latency = 3

4.b Resolving Data Hazards with Software Scheduling

```
         bne r1, r0, done
         lw r5, 0(r2)
         lw r6, 0(r3)
         addiu r8, r4, 4
         nop
         addu r7, r5, r6
         nop
         sw r7, 0(r8)
         ...
done:    addiu r10, r9, 1
```

By swapping the addu and addiu instructions, we can avoid 1 nop for the dependence between the addu and the lw instructions. Regardless of how the addiu and addu instructions are scheduled, there will always be a nop right before the sw instruction due to the dependence.

	Dynamic Transaction	Cycle 1	2	3	4	5	6	7	8	9	10	11	12	13
1	`bne r1, r0, done`	F	D	X0	X1	M	W							
2	`lw r5, 0(r2)`		F	D	X0	X1	M	W						
3	`lw r6, 0(r3)`			F	D	X0	X1	M	W					
4	`addiu r8, r4, 4`				F	D	X0	X1	M	W				
5	`nop`					F	D	X0	X1	M	W			
6	`addu r7, r5, r6`						F	D	X0	X1	M	W		
7	`nop`							F	D	X0	X1	M	W	
8	`sw r7, 0(r8)`								F	D	X0	X1	M	W

Figure 11: Software Scheduling Pipeline

Instruction 6 (addu)'s D stage depends on instruction 2 (lw)'s M stage and instruction 3 (lw)'s M stage. Instruction 7 (sw)'s D stage depends on instruction 4 (addiu)'s X1 stage and instruction 6 (addu)'s X1 stage.

4.c Resolving Data Hazards with Stalling

	Dynamic Transaction	Cycle 1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	`bne r1, r0, done`	F	D	X0	X1	M	W								
2	`lw r5, 0(r2)`		F	D	X0	X1	M	W							
3	`lw r6, 0(r3)`			F	D	X0	X1	M	W						
4	`addu r7, r5, r6`				F	D	D	D	X0	X1	M	W			
5	`addiu r8, r4, 4`					F	F	F	D	X0	X1	M	W		
6	`sw r7, 0(r8)`								F	F	D	X0	X1	M	W

Figure 12: Hardware Stalling Pipeline

Instruction 6 (addu)'s D stage depends on instruction 2 (lw)'s M stage and instruction 3 (lw)'s M stage.
Instruction 7 (sw)'s D stage depends on instruction 4 (addiu)'s X1 stage and instruction 6 (addu)'s X1 stage.

4.d New Stall Signal

```
ostall_load_use_X_rs_D
  =    val_D && rs_en_D && (val_X0 || val_X1) && (rf_wen_X0 || rf_wen_X1)
    && ((inst_rs_D == rf_waddr_X0) || (inst_rs_D == rf_waddr_X1))
    && ((rf_waddr_X0 != 0) || (rf_waddr_X1 != 0))
    && ((op_X0 == LW) || (op_X1 == LW))
ostall_load_use_X_rt_D
  =    val_D && rt_en_D && (val_X0 || val_X1) && (rf_wen_X0 || rf_wen_X1)
    && ((inst_rt_D == rf_waddr_X0) || (inst_rt_D == rf_waddr_X1))
    && ((rf_waddr_X0 != 0) || (rf_waddr_X1 != 0))
    && ((op_X0 == LW) || (op_X1 == LW))
ostall_alu_use_X_rs_D
  =    val_D && rs_en_D && val_X0 && rf_wen_X0
    && (inst_rs_D == rf_waddr_X0)
    && (rf_waddr_X0 != 0)
ostall_alu_use_X_rt_D
  =    val_D && rt_en_D && val_X0 && rf_wen_X0
    && (inst_rt_D == rf_waddr_X0)
    && (rf_waddr_X0 != 0)
stall_D
  =    val_D && !squash_D
    && ((ostall_load_use_X_rs_D || ostall_load_use_X_rt_D) ||
        (ostall_alu_use_X_rs_D || ostall_alu_use_X_rt_D))
```

4.e Resolving Control Hazards with Scheduling and Speculation

	Dynamic Transaction	Cycle									
		1	2	3	4	5	6	7	8	9	10
1	`bne   r1, r0, done`	F	D	X0	X1	M	W				
2	`lw    r5, 0(r2)`		F	D	X0	-	-	-			
3	`lw    r6, 0(r3)`			F	D	-	-	-	-		
4	`addu  r7, r5, r6`				F	-	-	-	-	-	
5	`addiu r10, r9, 1`					F	D	X0	X1	M	W

Figure 13: Squashing Pipeline

	Dynamic Transaction	Cycle									
		1	2	3	4	5	6	7	8	9	10
1	`bne   r1, r0, done`	F	D	X0	X1	M	W				
2	`lw    r5, 0(r2)`		F	D	X0	X1	M	W			
3	`lw    r6, 0(r3)`			F	D	-	-	-	-		
4	`addu  r7, r5, r6`				F	-	-	-	-	-	
5	`addiu r10, r9, 1`					F	D	X0	X1	M	W

Figure 14: Branch Delay Slot Pipeline