6.823 Computer System Architecture
VLIW, Vector Processing, and
Sequential Consistency
Problem Set #5

Spring 2002

Students are encouraged to collaborate in groups of up to 3 people. A group needs to hand in only one copy of the solution to a problem set. Homework assignments are due at the beginning of class on the due date. To facilitate grading, each problem must be stapled separately and your latest group number must appear on each problem (If your group has changed please indicate this and we will assign you a new group number). Homework will not be accepted once solutions are handed out. *It may be necessary to make certain assumptions in order to do the following problems. Be sure to explicitly state your assumptions in your write-ups.*

Problem 1: Predication

Ben Bitdiddle has the following C loop, which takes the absolute value of elements within a vector.

```
for (i = 0; i < N; i++) {
    if (A[i] < 0)
        A[i] = -A[i];
}
```

Problem 1.A

Translate Ben's loop into DLX, assuming elements in `A` and `N` are 32-bit signed integers. Initially, `R1` contains N and `R2` points to `A[0]`. You do not have to preserve the register values. Assume DLX has the standard single delay slot. Comment your code and optimize your code to improve performance but do not use loop unrolling or software pipelining.

What is the average number of cycles per element for this loop, assuming data elements are as equally likely to be negative as non-negative? Assume we are using the simple in-order five-stage fully bypassed DLX pipeline (L6-36) with memory made of MAGIC RAM.

Problem 1.B

Rewrite the assembly code by unrolling your original loop so that now two iterations of the original loop are performed before each backwards branch. What is the average number of cycles per element for large N? Does loop unrolling improve the number of cycles per iteration for the new loop? What are the tradeoffs for loop unrolling?

Problem 1.C

Alyssa P. Hacker comments that she can remove the data-dependent branches in the assembly code by using predication. She proposes a new set of predicated instructions for DLX, as follows:

1) Augment the ISA with a set of 32 predicate bits `P0-P31`.
2) Every standard non-control instruction now has a predicated counterpart, with the following syntax:

```
(pbit) DLX INSTRUCTION   ; execute the DLX instruction if
                         ; pbit is true
```

3) Include a set of compare instructions that conditionally set a predicate bit:

```
CMPLTZ pbit,reg     ; set pbit if reg < 0
CMPGEZ pbit,reg     ; set pbit if reg >= 0
CMPEQZ pbit,reg     ; set pbit if reg == 0
CMPNEZ pbit,reg     ; set pbit if reg != 0
```

Translate Ben's loop to DLX with the new predicated instructions. Make sure to comment your code to make it as understandable as possible. Try to optimize your code but do not use software pipelining or loop unrolling.

What is the average number of cycles per element for this new loop? Assume that the predicate-setting compares have single cycle latency (i.e., behave similarly to a regular ALU instruction including full bypassing of the predicate bit).

Problem 1.D

Now consider a predicated VLIW version of DLX that can issue two operations per cycle. Schedule the predicated loop body above to give the maximum execution rate without using loop unrolling or software pipelining. What is the average number of cycles per element? Assume the VLIW pipeline has the same five-stage pipeline structure as the original DLX but with two ALUs, memories with two read and write ports, and a register file with four read ports and two write ports. There is still a single delay slot of up to two operations.

Problem 1.E

Unroll the VLIW code to perform two iterations of the original loop before each backwards branch and schedule the code for best performance. What is the average number of cycles per element for large N?

Problem 1.F

If Ben knows the value of N so that he can completely unroll the loop with software pipelining, what would be the minimum average cycles per element? Explain. (Hint: You don't have to write the complete code schedule to answer this question.)

Problem 2: VLIW Programming

Ben Bitdiddle and Louis Reasoner have started a new company called Transbeta® and are designing a new processor named Titanium™. The Titanium processor is a single-issue in-order VLIW processor with:

- 2 load/store units. There is no cache. A load has a latency of 4 cycles but is fully pipelined.
- 1 integer ALU. single cycle
- 1 floating-point multiplier. 3 cycles, fully pipelined
- 1 floating-point adder. 2 cycles, fully pipelined
- 1 branch unit with no delay slots and 100% branch prediction accuracy
- 128 GPRs and 128 FPRs

A single Titanium instruction can issue to all the above units simultaneously. By definition, the operations in a Titanium instruction are independent. Every operation in a Titanium instruction reads the operands and issues simultaneously. Thus, if one operation is waiting for a result of a previous Titanium instruction, the entire Titanium instruction is stalled in the decode stage.

Everything is fully bypassed. Each functional unit has a dedicated writeback port, so there is never any contention. Writing to the same register multiple times in the same instruction is disallowed in the Titanium ISA. WAW hazards will also cause stalls. The Titanium ISA resembles DLX, except that there can be up to 6 instructions on each line separated by semicolons.

You have been hired to work on some hand-optimized math libraries. The most important of these is the dot-product, given by $\Sigma(X_n \times Y_n)$.

Ben has translated dot-product from DLX to the Titanium ISA

```
// R1 – pointer to X
// R2 – pointer to Y
// R5 - n
// R3 - temp
// F4 - temp
// F6 – result
    MOVI2FP F6,R0
loop:
    LF    F3,0(R1); LF   F4,0(R2); ADDI R5,R5,#-1
    MULTF F3,F3,F4; ADDI R1,R1,#4
    ADDF  F6,F6,F3; ADDI R2,R2,#4; BNEZ R5,loop
```

Each iteration takes 9 cycles but the program averages 8 cycles per vector element. Alyssa P. Hacker says that it can be done in 1 cycle per vector element for long vectors. Show Ben and Louis what the code should be. Louis isn't too bright so make sure your code is well commented.

Problem 3: Vectorizing `strcpy`

Ben Bitdiddle has bought a state-of-the-art vector machine, the Zirconium™, which has vector registers holding up to 32 elements, and has decided to vectorize his C library functions. As a starting point, he vectorizes the C function `memcpy`. The specification for `memcpy` is given as:

```
/* copy n words from ct to s, and return s. */
/* The actual C copies one byte at a time.  */
/* Our version copies one word at a time.   */
void *memcpy(void *s, void *ct, size_t n)
```

Ben implements `memcpy` in the following fashion, assuming `s`, `ct`, and `n` are in registers `R1`, `R2`, and `R3` respectively. Assume no delay slots.

```
    ADD    R5,R1,R0      ; store destination address in R5
    ADD    R4,R2,R0      ; store source address in R4
    ANDI   R6,R3,#31     ; N % 32
    MOVI2S VLR,R6        ; put length in vector length register
loop:
    LV     V1,R4
    SV     R5,V1
    SUB    R3,R3,R6      ; subtract elements
    SLLI   R6,R6,#2
    ADD    R4,R4,R6      ; bump source pointer
    ADD    R5,R5,R6      ; bump destination pointer
    ADDI   R6,R0,#32
    MOVI2S VLR,R6        ; reset full length
    BNEZ   R3,loop       ; any more to do?
```

Problem 3.A

The Zirconium processor has one load/store unit with a single lane that is fully pipelined with a latency of 10 cycles and dead time of 10 cycles. Instructions do not need to spend an extra cycle writing back values. All scalar instructions are executed on a separate 5-stage pipelined fully-bypassed datapath. Therefore, execution of scalar instructions and vector instructions maybe overlapped. How many cycles are required to copy each element when a very long memory vector is copied, i.e., in steady state?

Problem 3.B

Ben's next target is `strcpy`, defined as follows:

```
/* copy string ct to string s, including '\0' and return s */
/* The actual C copies one byte at time.                    */
/* Our version copies one word at a time.                   */
void *strcpy(void *s, void *ct)
```

The difference between `strcpy` and `memcpy` is that `strcpy` terminates when it sees the string terminating character `'\0'` while `memcpy` copies a given length.

Ben makes several attempts to vectorize the code, but gives up deciding that it is not vectorizable. Alyssa, however, informs Ben that this function can be vectorized using some additional types of vector instructions listed below:

`CLZM`	`R1,VM`	Counts the number of leading 0s in the vector-mask register `VM` and puts the result in `R1`. For example, if the contents of `VM` are `0001010...000`, `clzm R1,VM` puts 3 into `R1`.
`S—V` `S--SV`	`V1,V2` `F0,V1`	Compare the elements (`EQ`,`NE`,`GT`,`LT`,`GE`,`LE`) in `V1` and `V2`. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (`VM`). The instruction `S--SV` performs the same compare but using a scalar value as one operand.

Given the additional instructions, help Ben write vectorized code for the Zirconium processor. Assume `s` and `ct` are in register `R1` and `R2` respectively. The Zirconium processor does not have virtual memory and does not trap on memory protection violations on vector memory loads. Also assume that a string must be word-aligned. The terminating character must start at a word boundary and the remaining 3 bytes after the terminating character must be 0x0. (Hint: The ASCII value of `'\0'` is 0.)

Problem 3.C

Compare the performance of vectorized `memcpy` and vectorized `strcpy` with and without vector chaining. Specifically, how many cycles is required to transfer one element in steady state? Assume there is one vector compare unit with one lane and one cycle latency that compares whether two values are equal.

Problem 4: Performance of Vector Machines

The vector processor Germanium™ has a vector addition and a vector multiply unit with the following attributes:

1) Vector registers have 32 elements. The vector register file supports 2 read ports and 1 write port for each addition unit and multiplication unit.

2) The vector addition unit has a 2 cycle latency but is fully pipelined.

3) The vector multiplication unit has a 3 cycle latency but is fully pipelined.

4) The pipeline structure for vector machine is as follows:

   F D R X X ... X W
   (F: fetch, D: Decode, R: Vector register read, W: write back)

You are now given the following code:

```
I1: ADDV  V3,V2,V1
I2: ADDV  V4,V2,V1
I3: MULTV V5,V4,V3

Note: All vectors are 32 elements in length.
```

Problem 4.A

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, 2 cycle dead time, and no vector chaining. How many cycles does the given code take to execute? Count execution time as the number of cycles from when the first result is written to when the last result is written (inclusive).

For example, the pipeline diagram for `ADDV V3,V2,V1` and vector lengths of 24 elements, is shown below. Because we need to do 24 operations using 8 lanes, the vector register file should be read three times. X1 is the first stage of addition unit and X2 is the second. At cycle 6, the results of the first 8 operations are written back. This instruction takes 3 cycles to execute.

Time ⟶

Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8
F	D	R	X1	X2	W		
			R	X1	X2	W	
				R	X1	X2	W

Problem 4.B

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 8 lanes, no dead time, and vector chaining. Vectoring chaining is done through the register file. A vector unit can read an element from the register file in the same cycle it is being written back. How many cycles does the given code take to execute?

Problem 4.C

Draw a pipeline diagram of the Germanium processor running the given code, assuming it has 16 lanes, no dead time, and vector chaining. How many cycles does the given code take to execute?

Problem 5: Synchronization and Sequential Consistency

Cy D. Fect wants to run the following code sequences on processors P1 and P2.

Initially, `M[A]`, `M[B]`, `M[C]`, and `M[D]` are all `0`.

P1	P2
`ADDI R1,R0,#1`	`ADDI R1,R0,#1`
`SW A(R0),R1`	`SW B(R0),R1`
`LW R2,B(R0)`	`LW R2,A(R0)`
`SW C(R0),R2`	`SW D(R0),R2`

Problem 5.A

If Cy's code runs on a system with a sequentially consistent memory model, what are the possible results of execution? List all possible results in terms of values of `M[C]` and `M[D]`.

Problem 5.B

Assume now that Cy's code is run on a system that does not guarantee sequential consistency, but that memory dependencies are not violated for the accesses made by any individual processor. The system has a `MEMBAR` memory barrier instruction that guarantees the effects of all memory instructions executed before the `MEMBAR` will be made globally visible before any memory instruction after the `MEMBAR` is executed.

Add `MEMBAR` instructions to Cy's code sequences to give the same results as if the system were sequentially consistent. Use the minimum number of `MEMBAR` instructions.