6.823 Computer System Architecture

Introduction to ISA's
Problem Set #1

Spring 2002

Students are strongly 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. *Homework will not be accepted once solutions are handed out.*

As a reminder, students who have not signed up for the course mailing list should email TAs as soon as possible.

Students should read Chapters 1 and 2 of H&P for this problem set.

Problem 1: CISC versus RISC

In this problem, your task is to compare the instruction set architecture of the x86 with that of the DLX. The x86 is an extended accumulator, CISC architecture with variable length instructions. The DLX is a load-store, RISC architecture with fixed length instructions. Let us begin by considering the following C code:

```
for(i = 0; i < n; i++)
{
  sum += a[i];
}
```

Using gcc on a Pentium, the above loop compiles to the following x86 instruction sequence:

(On entry to this code, register `%ebx` contains `n`, and register `%ecx` contains `a`, and register `%edx` contains `sum`.)

```
     xorl %eax,%eax           # i = 0
     cmpl %ebx,%eax
     jge .L4
.L6:
     addl (%ecx,%eax,4),%edx
     incl %eax
     cmpl %ebx,%eax
     jl .L6
.L4:
```

The meanings and instruction lengths of the instructions used above are given in the following table. Registers are denoted with $R_{SUBSCRIPT}$, register contents with $<R_{SUBSCRIPT}>$.

Instruction	Operation	Length
addl (R_{BASE},R_{INDEX},SCALE),R_{SRC}	$R_{SRC} \leftarrow <R_{SRC}> +$ Mem[$<R_{BASE}>$+(SCALE×$<R_{INDEX}>$]	3 bytes
cmpl R_{SRC1}, R_{SRC2}	Temp $\leftarrow <R_{SRC2}> - <R_{SRC1}>$	2 bytes
incl R_{DEST}	$R_{DEST} \leftarrow <R_{DEST}> + 1$	1 byte
jge label	if (SF=OF) jump to the address specified by `label`	2 bytes
jl label	if (SF≠OF) jump to the address specified by `label`	2 bytes
xorl R_{DEST}, R_{SRC}	$R_{DEST} \leftarrow <R_{DEST}> \otimes <R_{SRC}>$	2 bytes

Notice that the jump instructions `jge` (jump if greater than or equal to) and `jl` (jump if less than) depend on SF and OF, which are status flags. Status flags, also known as condition codes, is analogous to the method of condition register used in the DLX architecture. Status flags are set by the instruction preceeding the jump, based on the result of the computation. Some instructions, like the `cmpl` (compare less than) instruction, perform a computation and set status flags, but do not return any result. The meanings of the status flags are given in the following table:

Name	Purpose	Condition Reported
OF	Overflow	Result exceeds positive or negative limit of number range
SF	Sign	Result is negative (less than zero)

Problem 1.A

For the above x86 assembly code, how many bytes of instructions need to be fetched if n = 10? Assuming 32-bit data values, how many bytes of data memory need to be fetched? Stored?

Problem 1.B

Translate each of the x86 instructions in the following table into one or more DLX instructions. You should use the minimum number of instructions needed. Use R1 for `n`, R2 for `a`, R3 for `i`, and R4 for `sum`. If needed, use R5 as a condition register, and R6, R7, etc., for temporaries. You should not need to use any floating point registers or instructions in your code.

x86 instruction	DLX instruction sequence
`xorl %eax,%eax`	
`cmpl %ebx,%eax`	
`jge .L4`	
`addl (%ecx,%eax,4),%edx`	
`incl %eax`	`ADDI R3,R3,#1`
`cmpl %ebx,%eax`	
`jl .L6`	

A description of the DLX instruction set architecture can be found in Chapter 2.8 of Hennessy & Patterson, *Computer Architecture A Quantitative Approach*.

Problem 1.C

Using your DLX translations from part 1.B, how many bytes of DLX instructions need to be fetched for n = 10? How many bytes of data memory need to be fetched? Stored? How does this compare to your answer for part 1.A?

Problem 1.D

Now provide a hand-optimized implementation of the same loop in DLX. How many bytes of instructions need to be fetched for n = 10? How many bytes of data memory need to be fetched? Stored? How does this compare to your answers for parts 1.A and 1.C?

Problem 2: Push and Pop

After learning about stack-based instruction sets in 6.823, Ben Bitdiddle thinks it would be a great idea to add push and pop to the DLX instruction set. Ben thinks that having stack instructions would make manipulating the call stack much easier. He decides to reserve register R29 for holding the stack pointer, and gives the following semantics for the push and pop instructions (stack grows down):

Instruction	**Operation**
`push` R_{SRC}	`Mem[<R29> - 4]` ← $<R_{SRC}>$ `R29 ← <R29> - 4`
`pop` R_{DEST}	R_{DEST} ← `Mem[<R29>]` `R29 ← <R29> + 4`

Problem 2.A

What modifications would need to be made to the basic DLX pipeline (given in Ch.3 of Hennessy & Patterson) to accommodate these extra instructions? What restrictions might you want to make on R_{DEST}?

Problem 2.B

With a 6-bit opcode, and a 5-bit field to encode the register, push and pop each take only 11 bits to encode. This means that 21 bits of the 32-bit instruction are wasted. Ben decides that it would be a better idea to move DLX from fixed-length to variable-length encoding to save some instruction memory, and make push and pop 11 bits long each. He mentions this idea to his friend, Alyssa P. Hacker. Alyssa vetoes this idea at once.

Why might it not be a good idea to make push and pop 11 bits long? What would be a better length? Assuming that we do not want to change the meanings of the other DLX instructions, what problems might we encounter? (Hint: Think about branch and jump instructions.)

Problem 2.C

In order to evaluate the effectiveness of the new instructions, Ben decides to consider typical stack manipulations. He adopts the following calling convention:

- The stack is grown downward and the current stack bottom is kept in the stack pointer sp (R29). The stack pointer points to the last full location on the stack.
- To call a subroutine, the caller creates a data structure on the stack to hold the arguments and sets sp to point to it. The first argument is lowest in memory. You should allocate a minimum of 16 bytes of argument space for any call, even if the arguments would fit in less.

- Any arguments allocated to the first 16 bytes of the argument structure are passed in registers. The caller should leave the first 16 bytes of the structure undefined. The four words of register argument values go in R4 through R7.
- Return values will be in register R2.
- R16 through R23, and R30 are callee-saved registers (a subroutine guarantees that these registers contain the same values at its exit as they did on entry).

In this problem, we will only deal with 32-bit integer arguments and 32-bit integer return values. Consider the following C function and its implementation in DLX:

```
int bar(int d, int e, int f, int g)
{
  int h = foo(&d, &e, &f);

  return (d+e+f+g+h);
}
```

```
bar:
        SUBI    sp,sp,#24   # reserve 6 words on the stack
        SW      16(sp),R16  # save callee-saved register R16
        ADD     R16,R7,R0   # put argument g in R16
        SW      24(sp),R4   # save argument d
        ADDI    R4,sp,#24   # put the address of d in R4
        SW      28(sp),R5   # save argument e
        ADDI    R5,sp,#28   # put the address of e in R5
        SW      32(sp),R6   # save argument f
        ADDI    R6,sp,#32   # put the address of f in R6
        SW      20(sp),R31  # save the contents of the link register
        JAL     foo         # jump and link to foo
        LW      R3,24(sp)   # restore d
        LW      R4,28(sp)   # restore e
        LW      R5,32(sp)   # restore f
        LW      R31,20(sp)  # restore the link register
        ADD     R3,R3,R4
        ADD     R3,R3,R5
        ADD     R3,R3,R16
        LW      R16,16(sp)  # restore callee-saved register R16
        ADD     R2,R3,R2    # put return value in R2
        ADDI    sp,sp,#24   # restore stack
        J       R31         # return to caller
```

How many bytes of instructions need to be fetched for the above implementation of bar?

Problem 2.D

Rewrite the above assembly, using the push and pop instructions for any stack loads and stores. How many bytes of instructions need to be fetched for your new implementation? How does this compare to your answer for part 2.C? What does this tell you about push and pop?

Problem 2.E

Ben is still thinking about how to save instruction memory, and decides to add two more instructions to the DLX instruction set, inspired by a similar set of instructions in the ARM instruction set architecture. They are the `STMFD` (store multiple) and `LDMFD` (load multiple) instructions, which allow multiple registers to be loaded or stored using a single instruction. The semantics of these instructions are given in the following table:

Instruction	**Operation**

STMFD R_{BASE},{register_list}	for each register $R_i \neq R0$ in register_list
	  Mem[$<R_{BASE}>$ - 4] $\leftarrow$ $<R_i>$
	  R_{BASE} $\leftarrow$ $<R_{BASE}>$ - 4

LDMFD {register_list},R_{BASE}	for each register $R_i \neq R0$ in register_list R_i $\leftarrow$ Mem[$<R_{BASE}>$] R_{BASE} $\leftarrow$ $<R_{BASE}>$ + 4

`register_list` is an in-order list of registers, from lowest-numbered to highest numbered (ex. {R1-R7,R9}).

Alyssa steps in once again and points out to Ben that the semantics of these instructions aren't quite right. What is the problem, and how can it be fixed?

Problem 2.F

Take your code from part 2.D and replace pushes and pops with store and load multiple instructions. Assuming we can encode a store or load multiple instruction in 32 bits, how many bytes of instructions need to be fetched for your new implementation? What if we need 64 bits? How does this compare to your answers for parts 2.C and 2.D? What does this tell you about store and load multiple instructions?

Problem 3: DSP Extension for DLX ISA

DLX-Rules, Inc. manufactures microprocessors based on the DLX ISA. The current generation DLX processor takes one CPU cycle to execute any instruction except load and store. Instructions can only be executed in program order. The processor does not have cache memory. Both load and store always take five CPU cycles.

The company recently found potential customers in real-time audio processing business and started to develop a new processor for audio processing applications. The core of the audio processing is filtering, which is performed by the following code.

C implementation

```
sum = 0;

for (i = 0; i < N; i++) {
     sum += a[i]*h[N-1-i];
}
```

Problem 3.A

Write DLX assembly code for this audio processing core that minimizes the number of instructions inside the loop. (Assume that R1 and R2 hold the pointers to array ‘a’ and ‘h’, respectively. Use registers R3, R4, and R5 for variable ‘sum’, ‘i’, and ‘N’. Each array element is 32 bits.)

Problem 3.B

For N = 100, how many instructions will be executed to complete the assembly code in 3.A? How many CPU cycles will it take? (All instructions other than load/store take one cycle, and load/store instructions take 5 cycles. The processor is NOT pipelined and executes one instruction at a time.)

Problem 3.C

Many customers aren’t satisfied with the performance of the current DLX processor and asked for a faster processor. Given the fact that the convolution is a very basic operation for various multimedia applications, DLX-Rules decided to have instructions for convolution and hardware support for them. The following two instructions are proposed.

Opcode	**Description**

MACC Rd, Rs1, Rs2, Im1, Im2	Rd ← <Rd> + (Mem[<Rs1>]×Mem[<Rs2>]),
	Rs1 ← <Rs1> + Im1,
	Rs2 ← <Rs2> + Im2
	All operands (Rs1, Rs2) are read before updated.

REPEAT Rs	Repeat the next instruction <Rs> times.

Table 3-1: DSP Instructions

DLX-Rules wants to minimize the hardware modifications and decides to support the given instructions using simple addition of translation logic to the existing DLX processor. The idea is to translate the new instructions to their corresponding instructions in the original DLX ISA dynamically in hardware. Once an instruction is read from the memory, hardware checks if the instruction is either MACC or REPEAT. If so, the instruction is replaced with the corresponding sequence of original DLX instructions, which are executed one by one.

Will this approach work for MACC and REPEAT without any other hardware modification? Write the corresponding DLX instruction sequence for each new instruction. If there is a problem, please clearly explain the problem and additional hardware support required for translation.

Problem 3.D

Write assembly code for the given audio processing core using the DSP instructions (first using only MACC, and then using both MACC and REPEAT). For N = 100, how many instructions and cycles does the program take to complete for each case? Is there any performance advantage of using new instructions when the instructions are implemented based on the translation in 3.C?

Problem 3.E

Ben, who's newly hired for the processor implementation of the DSP extension, thinks that the processor should have more aggressive hardware support to achieve high performance. He implemented specialized hardware for MACC, which performs two memory reads and two additions in parallel (5 cycles), then a multiplication and an addition sequentially (2 cycles). Now each MACC instruction takes 7 cycles to execute. For N = 100, how many cycles would it take to complete the program that uses both MACC and REPEAT?

How can we further improve the performance of this processor by modifying the specialized hardware described above? How many cycles would the program take?

Problem 4: Addressing Modes

Ben Bitdiddle notices that the only data-addressing mode for DLX is displacement addressing. After taking 6.823, Ben learns that the VAX ISA has more addressing modes than the DLX. In the mid 1970s, when the VAX was designed the prevailing philosophy was to create instruction sets that were close to programming languages to simplify compilers. Thus, the VAX provided a large set of addressing modes, which limited code size and reduced register usage.

Ben wants to know how additional addressing modes in DLX can minimize code size and register usage for the following code sequence.

```
for (i = 0; i < n; i++)
     sum += a[i];
```

Initially, R1 contains `n`, R2 contains `a`, R3 contains `i`, and R4 contains `sum`.

Problem 4.A

Looking at the optimized DLX assembly from question 1.D, how many temporary registers are required? For n = 10, how many dynamic instructions are executed?

Problem 4.B

Ben wants to know how adding indirect addressing-mode for arithmetic instructions to DLX affects his code. Rewrite the code sequence from 4.A, optimizing for static code space and register usage. The syntax for an add using indirect addressing-mode is:

```
ADD R4, (R1)    // R4 ← R4 + M[R1]
```

How many temporary registers are required? For n = 10, how many dynamic instructions would the code sequence take?

Problem 4.C

Next, Ben wants to see how adding autoincrement addressing-mode for arithmetic instructions to DLX affects his code. Rewrite the code sequence from 4.A, optimizing for static code space and temporary register usage. The syntax for an add using autoincrement addressing-mode is:

```
ADD R1, (R2)+    // R1 ← R1 + M[R2]; R2 ← R2 + d
                 // d is the size of an element
```

How many temporary registers are required? For n = 10, how many instructions would the code sequence take?

Problem 4.D

What changes to the 5-stage pipelined DLX machine will Ben need to make to add indirect addressing? What changes will he need to make to add autoincrement addressing? For both cases, what problems might Ben encounter?

Problem 5: Making Ends Meet on the EDSACjr

The first computer architects did not know exactly what was needed in their machines. They did know, however, that parts were expensive and unreliable. Thus, these pioneers developed architectures that minimized hardware while attempting to provide sufficient functionality to programmers. One of the first electronic computers, EDSAC, had a single accumulator and only absolute addressing of memory. Since one could reference memory only by an address listed explicitly within a program, self-modifying code was essential.

We now know that register based addressing and indirection make assembly level programming and compilation a lot easier. This lesson was learned, however, after programmers spent nearly five years programming absolute addressing machines. This problem gives us a flavor of EDSAC-style programming and its limitations. We will use an instruction set that gives the functionality of EDSAC without some of the more obscure opcodes. This instruction set, named the EDSACjr, is described in table 5-1. In the notation used in the table below, M[x] stands for the contents of the memory location addressed by x. Accum refers to the accumulator. (Accum) stands for the contents of the accumulator. $\leftarrow$ signifies that data is transferred (copied) from the location to the right of the $\leftarrow$ to the location on the left. The immediate variable n is an address or a literal depending on the context. The EDSACjr architecture allows programmers to put constants at any memory location when a program is loaded.

Opcode	**Description**	**Bit Representation**
ADD n	Accum $\leftarrow$ (Accum) + M[n]	00000 n
SUB n	Accum $\leftarrow$ (Accum) - M[n]	10000 n
STORE n	M[n] $\leftarrow$ (Accum)	00010 n
CLEAR	Accum $\leftarrow$ 0	00011 00000000000
AND n	Accum $\leftarrow$ (Accum) & M[n]	00100 n
SHIFTR n	Accum $\leftarrow$ (Accum) shiftr n	00101 n
SHIFTL n	Accum $\leftarrow$ (Accum) shiftl n	00110 n
BGE n	If (Accum) ≥ 0 then PC $\leftarrow n$	00111 n
BLT n	If (Accum) < 0 then PC $\leftarrow n$	01000 n
NOP	Wait one cycle	01001 00000000000
END	Halt machine	01010 00000000000

Table 5-1: The EDSACjr instruction set

The shifts are arithmetic shifts. All words are 16 bits long. As in EDSAC, instructions are encoded as integers. The first 5 bits are the opcode and the last 11 bits form the immediate field (an 11-bit immediate address addresses up to 2048 words (16-bit) of memory -- twice that of the real EDSAC). Integers are represented in 16 little Endian bits, the most significant bit being a sign bit.

Problem 5.A Writing Macros For Indirection

With only absolute addressing instructions provided by the EDSACjr, writing self-modifying code becomes unavoidable for almost all non-trivial applications. It would be a disaster, for both you and us, if you put everything in a single program. As a starting point, therefore, you are expected to write *macros* using the EDSACjr instructions given in table 5-1 to *emulate* indirect addressing instructions described in table 5-2. Refer to the supplement handout, *Using Macros*, for an explanation of macros. Then you can use these macros in your programs as if they were real instructions. Using macros may increase the total number of instructions that need to be executed because certain instruction level optimizations cannot be fully exploited. However, the code size *on paper* can be reduced dramatically when macros are used appropriately. This makes programming, debugging, and grading (!) much easier. **Write the macros for the following opcodes:**

Opcode	Description
ADDind *n*	Accum ← (Accum) + M[M[*n*]]
STOREind *n*	M[M[*n*]] ← (Accum)
BGEind *n*	If (Accum) ≥ 0 then PC ← M[*n*]
BLTind n	If (Accum) < 0 then PC ← M[*n*]

Table 5-2: Indirection Instructions

Problem 5.B Subroutine calling conventions

A possible subroutine calling convention for the EDSACjr is to place the arguments right after the subroutine call and pass the return address in the accumulator. The subroutine can then get its arguments by offset to the return address.

Describe how you would implement this calling convention for the special case of two arguments and one return value using the EDSACjr instruction set. What do you need to do to the subroutine for your convention to work? What do you have to do around the calling point? How is your result returned? You may assume that your subroutines are in set places in memory and that subroutines cannot call other subroutines. You are allowed to use the original EDSACjr instruction set shown in Table 5-1, as well as the indirection instructions listed in Table 5-2.

To illustrate your implementation of this convention, write a program for the EDSACjr to iteratively compute **minimum(n, array)**, where **n** is an integer and **array** is a pointer to an array of integers. **minimum(n, array)** returns the minimum of the first n array values. The details are given by the C code below. Assume the array is initialized with valid integer values and that 1 <= **n** <= length(array) in the initial call. Make **minimum** a subroutine. In a few sentences, explain how your convention could be generalized for subroutines with an arbitrary number of arguments and return values?

The following program defines the iterative subroutine **minimum** in C.

C Implementation

```
int  minimum(int n, int *array) {
     int i;
     int min = array[0];
     for(i = 1; i < n; i++) {
          if (array[i] < min)
               min = array[i];
     }
     return min;
}
```

Problem 5.C Subroutine Calls

Using the convention suggested at the beginning of Problem 5.B, can subroutine calls be made from within subroutines? Give specific examples of when it is okay and when the convention would break down.

Problem 5.D Recursive subroutines

Design a calling convention for *recursive* 2 arguments / 1 return value subroutines. Your convention should support subroutines that can call themselves. How are the arguments passed in and how is the result returned? Again, you may use the indirection instructions defined in Table 5-2.

To illustrate your implementation of this convention, write a program for the EDSACjr which recursively computes **minimum(n, array)**. Make **minimum** a subroutine. (The C code for a recursive version of **minimum** is given below). In a few sentences, explain how your convention would be generalized for procedures with a different number of arguments and a different number of returned values.

The following program defines the recursive subroutine **minimum** in C.

C implementation

```
int  minimum(int n, int *array) {
     int min = array[0];
     if (n>1) {
          min = minimum(n – 1, array);
          if (array[n-1] < min)
               min = array[n-1];
     }
     return min;
}
```