# 1 Instructions

You may work in **pairs** with a **partner** on this project if you wish or you may work **alone**, although I **strongly encourage** you to work with a partner. If you do work with a partner, only **one** of you should **submit** the project to Blackboard. Make sure to put **both** of your names in the **source code** files and **name** the **tarball** following the instruction of **Section 4**. You will each **earn** the **same** number of points. **Section 4** describes **what** to submit and by **when**; **read it now**.

# 2 Lab Project Objectives

1. Use the **GCC C++ compiler** and a small subset of compiler options (-c, -D -g, -o, -O0, -Wall) to compile a C++ program.
2. Use the Linux **make** command and a **make file** to automate the building of a project.
3. Write a complete C++ program involving **multiple classes** in **multiple source code files**.
4. Write and include **header files**. Understand how to use **#ifndef ... #endif** to prevent multiple header file inclusion.
5. Write C++ **class declarations** and **definitions**, including **constructors**, **destructors**, and **accessor**/**mutator** functions.
6. Write **overloaded functions** and **overload operators**.
7. Use **const** correctly.
8. Instantiate objects and call member functions. Pass parameters by-value and by-reference.
9. Effectively use **reference variables**.
10. Use the **bit-manipulation operators**.
11. Use **I/O streams** to read and write **text** and **binary files**.
12. Perform **formal testing** to document program correctness.

# 3 K1 Microcontroller Simulator

Your lab project is to write an **assembler** for the K1 microcontroller. An assembler is a program which translates code written in the assembly language of a microprocessor into equivalent binary machine language. Read the following sections for a discussion of the K1 microcontroller and the assembly language.

## 3.1 A K1 Microcontroller-Based Computer System

The **K1** is an extremely RISC 32-bit microcontroller architecture, supporting only 18 hardware instructions[1]. The instructions are discussed in Section 3.3. Reference Figure 1. The K1 contains two general-purpose 32-bit registers (A and B), one 32-bit program counter register (PC), one 32-bit stack pointer register (SP), and one 32-bit instruction register (I). A **word** in the K1 system is 32-bits. The RAM contains one megawords, i.e., there are $2^{20}$ RAM cells with addresses 0 through $2^{20}-1$ (in hex, 00000-FFFFF), and each RAM cell stores one 32-bit word. The RAM contains both data and program instructions. Only one program may be loaded into RAM at a time. Data and instructions may be located anywhere in the RAM, with the exception of the stack space. The stack space starts at the top of RAM, i.e., at system initialization, the SP register is initialized to FFFFF. The stack grows downward. The stack size is limited only by the RAM size and the memory locations consumed by data and instructions. At startup, the PC register is initialized to 00000. The K1 is a little-endian processor.

A complete K1 microcomputer system consists of the K1 microcontroller, a computer terminal, and a permanent data store. The terminal consists of a keyboard and a display screen. These external devices are accessed over a port-based I/O bus. Port 0 interfaces to the display; port 1 to the keyboard, and port 2 to the data store.

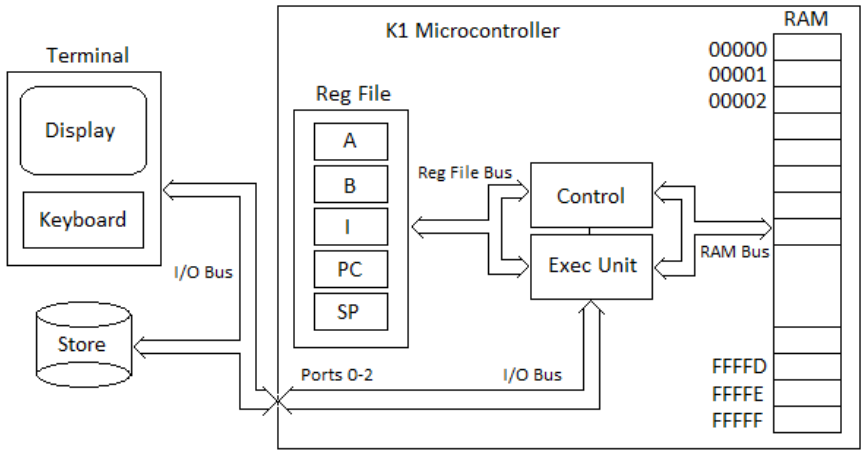


**Figure 1. Block Diagram of a K1-based Microcomputer System**

1 I made all this up. There is no such processor. However, it would be a good project to write a C++ simulator which simulates this system.

## 3.2 The K1 Execution Cycle

When a program is loaded into RAM, the microcontroller begins performing the following execution cycle,

1. *Initialize PC*: PC ← *initial value loaded from binary* (see Section 3.6)
2. *Initialize SP*: SP ← FFFFF
3. *Instruction Fetch Step*: PC is sent to RAM, and the instruction at PC is sent back to the control which writes it to the I register.
4. *Instruction Decode Step*: The instruction in I is decoded by the control to determine the opcode.
5. *Instruction Execution Step*: The opcode is sent to the execution unit. The execution unit executes the instruction, fetching operands from RAM or writing data to RAM if necessary.
6. *PC Update Step*: If the instruction is not a branch, PC is incremented. If the instruction is a branch, and the branch is taken, then PC is updated with the address in RAM of the instruction at the branch target.
7. Go to Step 3.

## 3.3 K1 Instruction Set

The K1 has 18 instructions. Each instruction is described below.

**Definitions**

| | |
|---|---|
| & | The address of the current instruction. |
| ~*x* | The one's complement of *x*. |
| ¬*x* | NOT *x*. |
| *addr* | An unsigned 20-bit RAM address encoded in bits 19:0 of the LD and ST instructions. |
| *imm* | A 26-bit two's complement immediate data value encoded in bits 25:0 of the LDI instruction. |
| *label* | A 20-bit RAM address encoded in bits 19:0 of the branch instructions. |
| *reg* | Either the A or B register. |
| *reg-other* | If *reg* is the A register, then *reg-other* is the B register. If *reg* is the B register, then *reg-other* is the A register. |
| *var* | A variable allocated in the .DATA segment. |
| *var-addr* | The address of variable *var*. |
| A | The A register. Encoded as 0 in the instruction. |
| Addr(*var*) | The address in memory of variable *var*. |
| B | The B register. Encoded as 1 in the instruction. |
| I | The I register. |
| PC | The PC register. |
| Port[*x*] | I/O port *x*. |
| RAM[*x*] | Memory address *x* in the RAM. |
| RTL | Register-transfer logic (RTL) equations, which specify what the execution unit must do. |
| SignExtend | Bit 25 of *imm* in the LDI instruction is copied to bits 31:26 of the destination register. |
| SP | The SP register. |

### 3.3.1 Load

| | |
|---|---|
| Mnemonic: | LD |
| Assembly syntax: | LD *reg var* |
| RTL: | *reg* ← RAM[*var-addr*] |
| Opcode: | 00000 |
| Encoding: | *opcode*[31:27] *reg*[26] *unused*[25:20] *var-addr*[19:0] |
| Description: | Loads a 32-bit value from the RAM into register A or B. |

### 3.3.2 Load Immediate

| | |
|---|---|
| Mnemonic: | LDI |
| Assembly syntax: | LDI *reg imm* |
| RTL: | *reg* ← SignExtend(*imm*) |
| Opcode: | 00001 |
| Encoding: | *opcode*[31:27] *reg*[26] *imm*[25:0] |
| Description: | Loads a 26-bit two's complement immediate into register A or B. The *imm* value is sign-extended. |

### 3.3.3 Load Address

| | |
|---|---|
| Mnemonic: | LDA |
| Assembly syntax: | LDA *reg var* |
| RTL: | *reg* ← Addr(*var*) |
| Opcode: | 00010 |
| Encoding: | *opcode*[31:27] *reg*[26] *unused*[25:20] *addr*[19:0] |
| Description: | Loads the address of variable *var* into register A or B. |

### 3.3.4 Store

Mnemonic: ST
Assembly syntax: ST *var reg*
RTL: RAM[*var-addr*] ← *reg*
Opcode: 00011
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:20] *var-addr*[19:0]
Description: Stores a 32-bit value from the A or B register into RAM at address *addr.*

### 3.3.5 Add

Mnemonic: ADD
Assembly syntax: ADD *reg*
RTL: *reg* ← *reg* + *reg-other*
Opcode: 00100
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Adds A and B and places the sum in A (or adds B and A and places the sum in B).

### 3.3.6 Negate

Mnemonic: NEG
Assembly syntax: NEG *reg*
RTL: *reg* ← ~*reg*
Opcode: 00101
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Negates the contents of register A or B. This is also called the one's complement.

### 3.3.7 Rotate Left

Mnemonic: ROL
Assembly syntax: ROL *reg*
RTL: *reg*[*i*+1] ← *reg*[*i*] for *i* = 30..0; (2) *reg*[0] ← *reg*[31]
Opcode: 00110
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Rotates the bits in register A or B one position to the left. The msb is rotated in to become the lsb.

### 3.3.8 Logical NOR

Mnemonic: NOR
Assembly syntax: NOR *reg*
RTL: *reg* ← ¬(*reg* ∨ *reg-other*)
Opcode: 00111
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Sets A to A NOR B (or B to A NOR B).

### 3.3.9 Push

Mnemonic: PUSH
Assembly syntax: PUSH *reg*
RTL: (1) SP ← SP – 1; then (2) RAM[SP] ← *reg*
Opcode: 01000
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Pushes the contents of register A or B so it becomes the top value on the stack.

### 3.3.10 Pop

Mnemonic: POP
Assembly syntax: POP *reg*
RTL: (1) *reg* ← RAM[SP]; then (2) SP ← SP + 1
Opcode: 01001
Encoding: *opcode*[31:27] *reg*[26] *unused*[25:0]
Description: Pops the top value from the stack and places it in the A or B register.

### 3.3.11 Branch

Mnemonic: BR
Assembly syntax: BR *label*
RTL: PC ← *label*
Opcode: 01010
Encoding: *opcode*[31:27] *unused*[26:20] *label*[19:0]
Description: The next instruction to be executed is at memory address *label*.

### 3.3.12 Branch on Equal

Mnemonic: BEQ
Assembly syntax: BEQ *label*
RTL: if A = B then PC ← *label* else PC ← PC + 1
Opcode: 01011
Encoding: *opcode*[31:27] *unused*[26:20] *label*[19:0]
Description: If the A and B registers are equal, the next instruction to be executed is at memory address *label*.

### 3.3.13 Branch on Less Than

Mnemonic: BLT
Assembly syntax: BLT *label*
RTL: If A < B then PC ← *label* else PC ← PC + 1
Opcode: 01100
Encoding: *opcode*[31:27] *unused*[26:20] *label*[19:0]
Description: If the A register is less than the B register, the next instruction to be executed is at memory address *label*.

### 3.3.14 Branch to Subroutine

Mnemonic: BSUB
Assembly syntax: BSUB *label*
RTL: (1) SP ← SP − 1; then (2) RAM[SP] ← PC + 1; then (3) PC ← *label*
Opcode: 01101
Encoding: *opcode*[31:27] *unused*[26:20] *label*[19:0]
Description: Branches and begins executing instructions in a subroutine.

### 3.3.15 Return from Subroutine

Mnemonic: RET
Assembly syntax: RET
RTL: (1) PC ← RAM[SP]; then (2) SP ← SP + 1
Opcode: 01110
Encoding: *opcode*[31:27] *unused*[26:0]
Description: Returns from a subroutine to the instruction following the original BSUB.

### 3.3.16 Port Input

Mnemonic: IN
Assembly syntax: IN *reg port*
RTL: *reg* ← Port[*port*]
Opcode: 01111
Encoding: *opcode*[31:27] *reg*[26] *unused*[24:1] *port*[0]
Description: Reads a word from an input port into register A or B.

### 3.3.17 Port Output

Mnemonic: OUT
Assembly syntax: OUT *port reg*
RTL: Port[*port*] ← *reg*
Opcode: 10000
Encoding: *opcode*[31:27] *reg*[26] *unused*[24:1] *port*[0]
Description: Writes a word from register A or B to an output port.

### 3.3.18 Halt

Mnemonic: HALT
Assembly syntax: HALT
RTL: PC ← PC
Opcode: 10001
Encoding: *opcode*[31:27] *unused*[26:0]
Description: Halts the processor. It basically goes into an infinite loop.

## 3.4 Pseudoinstructions

For Lab Project 5, pseudoinstructions have been eliminated.

## 3.5 Example Program

This section presents an example program which computes the sum of the first *n* positive integers. Pseudocode,

```
Sum ← 0
i ← 1
While i ≤ n Do
  Sum ← Sum + i
  i ← i + 1
End While
```

The K1 assembly language code,

```
; Defines a data segment. The data segment is located at memory address 512₁₀ = 00200h.

.DATA 512
$sum          0
$i            0
$n            3

; Defines a text (code) segment. The text segment is located at memory address 1024₁₀ = 00400h.

.TEXT 1024
              LDI     %A 0              ; %A <- 0
              ST      $sum %A           ; $sum <- 0
@loop         LDI     %B 1              ; $A <- $A + 1
              ADD     %A
              ST      $i %A             ; $i <- %A
              LD      %B $n             ; %B <- n
              BLT     @loop1            ; If $i < $n keep looping
              BEQ     @loop1            ; If $i = $n keep looping
              BR      @endloop          ; i > $n so drop out of loop
@loop1        LD      %B $sum           ; %B <- $sum
              LD      %A $i             ; %A <- $i
              ADD     %B                ; %B <- $sum + i
              ST      $sum %B           ; $sum <- $sum + i
              BR      @loop             ; Continue looping as long as $i ≤ $n
@endloop      HALT
```

## 3.6 Binary File Format

The output from the assembler is a binary, i.e., a file containing machine language code which can be loaded into RAM and executed. The binary file format consists of a 16-byte **header** followed by **two** variable-sized **segments**. The header stores information about the binary which is needed by the **loader** to load a program into RAM for execution. The format of the header is,

| Byte Offset | Contents | Description |
|---|---|---|
| 00–04h | 4B 31 42 49 4E | The text string "K1BIN". These are known as **magic numbers**. |
| 05-08h | *init-pc* | The initialization value for the PC register. An unsigned 32-bit integer. |
| 09-0Fh | *unused* | For future use. |

A segment contains either data or text (code). The format of a data segment is,

| Byte Offset | Contents | Description |
|---|---|---|
| 00h | *type* | The type of the segment (0 = data) |
| 01–04h | *size* | The size, in bytes, of the complete segment. |
| 05–08h | *address* | The location in RAM where the segment is to be loaded. |
| 09–(*size–1*) | *contents* | The contents of the data segment. Note that all global variables are initialized in the segment. |

The format of a text segment is,

| Byte Offset | Contents | Description |
|---|---|---|
| 00h | *type* | The type of the segment (1 = text) |
| 01–04h | *size* | The size, in bytes, of the complete segment. |
| 05–08h | *address* | The location in RAM where the segment is to be loaded. |
| 09–(*size–1*) | *contents* | The contents of the text segment. |

The first segment is at offset 10h from the beginning of the binary. For the example program of Section 3.5, assume the data segment is located in memory at address $512_{10}$ = 00200h (variable *sum* is at 00200h, variable *i* at 00201h, and variable *n* at 00202h) and the text segment is at address $1024_{10}$ = 00400h. The assembly language program, with expanded pseudoinstructions and instruction encodings is,

```
00400h              LDI       %A, 0       Encoding: 00001 0 00000000000000000000000000   = 08000000
00401h              ST        $sum %A     Encoding: 00011 0 000000 0000000001000000000  = 18000200
00402h  @loop       LDI       %B 1        Encoding: 00001 1 00000000000000000000000001   = 0C000001
```

```
00403h              ADD       %A          Encoding: 00100 0 00000000000000000000000000   = 20000000
00404h              ST        $i %A       Encoding: 00011 0 00000000000000000001000000001   = 18000201
00405h              LD        %B $n       Encoding: 00000 1 000000 00000000001000000010  = 04000202
00406h              BLT       @loop1      Encoding: 01100 0000000 00000000010000001001   = 60000409
00407h              BEQ       @loop1      Encoding: 01011 0000000 00000000010000001001   = 58000409
00408h              BR        @endloop    Encoding: 01010 0000000 00000000010000001110   = 5000040E
00409h  @loop1      LD        %B $sum     Encoding: 00000 1 000000 00000000001000000000  = 04000200
0040Ah              LD        %A $i       Encoding: 00000 0 000000 00000000001000000001  = 00000201
0040Bh              ADD       %B          Encoding: 00100 1 00000000000000000000000000   = 24000000
0040Ch              ST        $sum %B     Encoding: 00011 1 000000 00000000001000000000  = 1C000200
0040Dh              BR        @loop       Encoding: 01010 0000000 00000000010000000010   = 50000402
0040Fh  @endloop    HALT                  Encoding: 10001 00000000000000000000000000     = 88000000
```

If this assembly language program is stored in a file named **sum.s**, then invoking the K1 assembler on **sum.s** will produce a binary named **sum,**

```
$ k1as sum.s            assembles sum.s to produce a binary named sum
$ k1as sum.s -o foo     assembles sum.s to produce a binary named foo
```

The binary produced by the assembler will be 6Ah = $106_{10}$ bytes in size and the contents will be,

| Byte Offset | Contents | Description |
|---|---|---|
| 00–04h | 4B 31 42 49 4E | The text string "K1BIN". |
| 05–08h | 00 04 00 00 | PC is to be initialized to 00400h. |
| 09–0Fh | 00 00 00 00 00 00 00 | Unused. Should be set to zeros. |
| 10h | 00 | Data segment. |
| 11–14h | 15 00 00 00 | Size, in bytes, of the data segment ($21_{10}$ bytes). |
| 15–18h | 00 02 00 00 | The data segment is to be loaded into RAM at address 00200h. |
| 19–1Ch | 00 00 00 00 | Variable *sum* (initialized to 0) is at 00200h. |
| 1D–20h | 00 00 00 00 | Variable *i* (initialized to 0) is at 00201h. |
| 21–24h | 03 00 00 00 | Variable *n* (initialized to 3) is at 00202h. |
| 25h | 01 | Text segment. |
| 26–29h | 45 00 00 00 | Size, in bytes, of the text segment ($69_{10}$ bytes). |
| 2A–2Dh | 00 04 00 00 | The text segment is to be loaded into RAM at address 00400h. |
| 2E–31h | 00 00 00 18 | LDI %A 0 |
| 32–35h | 00 02 80 01 | ST sum %A |
| 36–39h | 01 00 00 0C | LDI %B 1 |
| 3A–3Dh | 00 00 00 20 | ADD %A |
| 3E–41h | 01 02 00 18 | ST $i %A |
| 42–45h | 02 02 00 04 | LD %B $n |
| 46–49h | 09 04 00 60 | BLT & + 3 |
| 4A–4Dh | 09 04 00 58 | BEQ & + 2 |
| 4E–51h | 0E 04 00 50 | BR & + 6 |
| 52–55h | 00 02 00 04 | LD %B $sum |
| 56–59h | 01 02 00 00 | LD %A $i |
| 5A–5Dh | 00 00 00 24 | ADD %B |
| 5E–61h | 00 02 00 1C | ST $sum %B |
| 62–65h | 02 04 00 50 | BR & – 11 |
| 66–69h | 00 00 00 88 | HALT |

## 3.7 Programming-Related Notes to Make Your Life Easier

1. You may assume that each line of text in the assembly language file will be 128 characters or fewer in length.
2. You may assume that there will be **only one** data segment in the source code and binary files. It may be empty.
3. You may assume that there will be **only one** text segments in the source code and binary files. It may be empty.
4. As a consequence of 2 and 3, you may assume that a binary will not contain **exactly two** segments total.
5. You may assume that there will be no more than 1024 variable definitions in the data segment.
6. You may assume that there will be no more than 1024 instructions in the text segment.
7. The .DATA segment may appear above or below the .TEXT segment in the source code file.
8. You may assume that the source file will not contain any syntax errors.
9. You may assume that the directives .DATA and .TEXT wil be written in all uppercase.
10. You may assume that instructons mnemonics will be written in all uppercase.
11. You may assume that registers %A and %B will be preceded by a % and will be written in uppercase.
12. You may assume that variable names will be preceded with a $ and will be in all lowercase letters and may contain digits.
13. You may assume that all numbers are valid decimal integers.

14. You may assume that the command line will contain a valid source filename in the format **name.s**.
15. The -o command line option may/may not be present. If it is present, you may assume that it will be followed by a valid filename.
16. You may assume that no other options or arguments will be on the command line.

### 3.8 Testing

Test your assembler on the program of Section 3.5. I would recommend you test it on other testing source code files as well.

## 4 What to Submit for Grading and by When

When your program is complete, type the following commands to "clean" the source code directory and create a bzipped tarball named **cse220-p05-*lastname*.tar.bz2,**

```
~ $ cd cse220/p05/k1as
~/cse220/p05/k1as $ make clean
~/cse220/p05/k1as $ cd ..
~/cse220/p05 $ tar cvjf cse220-p05-lastname.tar.bz2 k1as
```

Where *lastname* is your surname (or your first name if you do not have a surname). If you work with a partner, be sure to put both of your names in the source code files (in the AUTHORS section of the header comment blocks) and name your tarball **cse220-p05-*last name1-lastname2*.tar.bz2**.

Submit a working make file so the TA can build and test your project by typing **make clean** followed by **make**.

Submit this tarball to Blackboard using the lab project submission link before the deadline. The **deadline is Wed 7 Dec 2011 at 4:00am** (that's four o'clock in the morning on Wednesday, or very late Tuesday night for those of you who will be attempting to pull an all-nighter). C**onsult the online syllabus for the late and academic integrity policies.**