# Computer Organization & Architecture
# Program 3- B17 Simulator

By Zachary Owen
With ‘Special’ thanks to Elliott Rarden, Allison Bodvig, and Anthony Morast

# Introduction

The B17 Simulator is a 24-bit word-addressable accumulator architecture. It supports up to 64 instructions, up to four addressing modes, and $2^{12}$ (4096) words of memory.Main memory consists of 4,096 words, each of which is 24 bits wide. Addresses are 12 bits wide, and range from `000` through `FFF`. Within theword, bits are numbered from right to left, with bit 0 the least significant bit and bit 23 the most significant bit. The memory interface transfers one word of data at a time. Communication with memory is via two registers, MAR and MDR.

The input file consists of three separate objects -memory address, number of instructions and instructions. The first vale read from a input file is the memory address followed by the number of instructions.Once both these values are read in ,the firstinstruction is read in the given memory address and and the next subsequent instructions are saved in the subsequent memory addresses until we are done with all the given nmber of instructions.

For every instruction read in, a trace line is printed out on the console window. The trace line contains the following information-the address of the instruction(three hex digits), the instruction itself(six hex digits),the instruction mnemonic(four characters), the EA being used by this instruction if it uses a memory address(three hex digits) or “IMM” if it specifies an immediate operand, or three spaces and lastly the contents of AC(accumulator) and four index registers(X0,X1,X2,X3) after the instruction has been executed.The program only stops running when a HALT instruction is executed,an undefined OPCODE is encountered, an unimplemented OPCODE is encountered, an invalid or illegal addressing mode is specified for execution or an unimplemented addressing mode is encountered.

## Registers and Busses:

**Figure 1.1: Registers**

| Item | Size | Name | Contents/Usage |
|---|---|---|---|
| MAR | 12 | Memory Address Register | Address of the memory location which is to be loaded from or stored into. |
| IC | 12 | Instruction Counter | Address of the next instruction to be fetched, decoded, and executed. |
| X*n* | 12 | Index Registers | Four registers (X0-X3); contain values to be used in calculating memory addresses. |
| ABUS | 12 | Address Bus | Used when addresses are to be moved. |
| MDR | 24 | Memory Data Register | Data to be written into, or data most recently read from, memory. |
| AC | 24 | Accumulator | The accumulator register. |
| ALU | 24 | Arithmetic-Logic Unit | Performs computations. |
| IR | 24 | Instruction Register | Instruction being decoded and executed. |
| DBUS | 24 | Data Bus | Used when data and instructions are to be moved. |

The Registers and Busses are implimented as global externs defined in “globals.cpp” and initialized in “b17.cpp”. In general the communication is only allowed between a bus and a registers, or a register and a bus. The only exception is that the Index Registers are alowed to communicate directly with the Accumulator. There is also a Hidden register called ‘EA’ which is used to hold the calculated Effective address calculated each cycle, and is helpful in printing the Trace information each cycle.

## Addressing Modes:

The B17 Simulaton is allowed to access memory for operands in 5 legal ways. These way’s are; Direct, Immediate, Indexed, Indirect, and Indirect Indexed. The accessing is handled by the getVal() and getEA() functions (found in operators.cpp). Operands use the addressing modes in different ways but in general most simple use getVal() inorder to get the information sored at that place in memory. OpST(), the store instruction uses getEA() so that it can know where it will eventually store its data.

| Bits 5-2 | Name | Interpretation |
|---|---|---|
| 0000 | Direct | EA is the contents of the address field. |
| 0001 | Immediate | There is no EA - the operand data is the sign-extended address field contents. |
| 0010 | Indexed | EA is the sum of the address field and the specified index register. |
| 0100 | Indirect | Address field contains the address of an indirect word in memory; the EA is the upper 12 bits of that word. |
| 0110 | Indexed Indirect | Perform indexing; the result is the address of an indirect word, whose upper 12 bits is the EA. |
| other | Illegal | Causes an ``illegal addressing mode'' error |

**Figure 1.2: Addressing Modes**

The following table gives you a look about each instruction and the legal addresing modes for it.

All –The following instructions can work along with any of the addressing modes.These instructions are : LD, ADD, SUB, AND, OR and XOR.

Direct –The following instructions can work along with only direct addressing mode. These instructions are : STX and EMX.

Immediate,Direct -The following instructions can work along with only direct addressing mode

and immediate addressing mode. These instructions are : LDX, ADDX and SUBX.

All except Immediate –The following instructions can work along with any of the addressing modes but the immediate addressing mode.These instructions are : ST, EM, J, JZ, JN and JP.

Ignored –The following instruction don't takethe addressing mode into account while execting.These instructions are : HALT, NOP, CLR, COM and CLRX

## Opcodes:

**Figure 1.3 Opcode Diagram**


**Bits 23-12** **Bits 11 – 6** **Bits 5-0**

The basic unit of computation in our machine is the OP code. OPCODE's are represented by uint32_t type integers and simulate 24 bit numbers. An OPCODE is compiresed of 3 parts, the Address (bits 23-12) which is a 3 HEX digit number usually representing a place in RAM, an Operator (bits 11 -6)

which Represent indexes on the **Figure 1.4.** And is used to tell the B17 which operation to perform. There are 64 possible instructions although only 22 are presently in use. Attempting to Use an undefined operation will cause the Simulation to HALT and ERR. And The final part is the Addressing Mode (bits 5-0) discussed previously.

**Figure 1.4 Operator Graph**

| | Bits 11-10 | | | |
|---|---|---|---|---|
| **Bits 9-6** | **00** | **01** | **10** | **11** |
| **0000** | HALT | LD | ADD | J |
| **0001** | NOP | ST | SUB | JZ |
| **0010** | ? | EM | CLR | JN |
| **0011** | ? | ? | COM | JP |
| **0100** | ? | ? | AND | ? |
| **0101** | ? | ? | OR | ? |
| **0110** | ? | ? | XOR | ? |
| **0111** | ? | ? | ? | ? |
| **1000** | ? | LDX | ADDX | ? |
| **1001** | ? | STX | SUBX | ? |
| **1010** | ? | EMX | CLRX | ? |
| **1011** | ? | ? | ? | ? |
| **1100** | ? | ? | ? | ? |
| **1101** | ? | ? | ? | ? |
| **1110** | ? | ? | ? | ? |
| **1111** | ? | ? | ? | ? |

## Halt States:

Inorder to simulate a real computer better Several **HALT** states have been introduced to help with debugging. If the program ever enters a state where it cannot proceed due to poor data or is given the HALT OPCODE the machine will print “Machine Halted - “ followed by one of the messages from Figure 1.5. These messages can give key information as to why your simulation is running about as well as Windows ME.

| Message | Reason | Action |
|---|---|---|
| `HALT instruction executed` | A HALT instruction was executed. | Stop the simulation. |
| `undefined opcode` | An opcode which is not defined (see above) was encountered. | Print `????` for the mnemonic in the trace; perform as a `NOP`; stop the simulation. |
| `unimplemented opcode` | An opcode which is not implemented (see above) was encountered. | Print the mnemonic in the trace; perform as a `NOP`; stop the simulation. |
| `illegal addressing mode` | An invalid mode was specified, or a mode which is illegal for this instruction. | Print `???` for the memory address in the trace; perform as a `NOP`; stop the simulation. |
| `unimplemented addressing mode` | An addressing mode which is not implemented (see above) was encountered. | Print `???` for the memory address in the trace; perform as a `NOP`; stop the simulation. |

**Figure 1.5 Halt Messages**

# Implimentation

## b17.cpp:

### Functions:

**uint32_t countNumberOfLinesInFile(char* filename):** This function counts and returns the number of lines in the file by opening steping through line by line , closing the file then exiting.

**void decodeToInstruction(instruction &instruct):** This function accesses the 32bit integer IR as one of the global variables and parces its pieces into instruct in the proper way. Only the lower 24 bits are used as that is the size of an OPCODE.

**void runInstruction(instruction $instruct ):** This function calls the function pointer array that holds the functions containing the operators. (see operators.c)

**int32_t loadProgram(char* filename, uint32_t numLines):** This function runs through a given file and while detecting error places the numbers into the RAM array

**void startProgram(uint32_t addresssOfFirstInstruction ):** This fucntion sets up the main loop places the first opcode into memory and begins the main loop.

**void printTrace(instruction instruct, uint32_t address, uint32_t instruction, ostream &out):** Part of the specifiation is that every cycle the program is to print a trace consisting of the values in each register along with the line of code being run, the value at that line, the effective address being used and the Mnmonic for the operation.

**void printAddressMode( instruction instruct ):** Helper function that properly prints out what addressing mode each operator is using.

**void haltingOutput( ):** Function that handles output incase of crash or halting.

## Operators.cpp

**void RunMEM( char MemAcc ):** Since we are trying to simulate a real machine this function performs a memory access as thought we were really using the B17. For Loading the address to load from must be present on the ABUS, and for Storing the address must be on the ABUS and the Value to store must be on the DBUS. It's operand represents what type of memory access to do and has only 2 possible values _WRITEMEM and _READMEM which are 'd in operators.cpp

**void RunALU( char operand):** This function performs similar porpose as RunMEM() except it simulates an ALU operation in all cases except operations using the index registers the operands are first the accumulator and second what is present on the DBUS. Return values are placed in the ALU register and can be fetched from there. Its operand and again like RunMEM s that determine what operation to use they are.. _NOP_ , _ADD_ , _SUB_ , _COM_ , _CLR_ , _AND_ , _OR_ , _XOR_, _ADDX_, _SUBX_,_CLRX_ and the meanings should be evident.

**int32_t getEA( instruction &instruc, char allowed ):** This function Fetches the would be EA(Effective Address) for the operand instruct from RAM. This function will set error flags in the PSR if it is unable use the access mode provided in allowed. The operand 'allowed' should be ORed values from the address mode s in operators.cpp they are identified by the format _AM_XXX and are _AM_DIR , _AM_IMM, _AM_IND, _AM_IDR, _AM_IDI , _AM_ALL, standing for DIRect , IMMediate , INDexed, InDiRect, InDirect Indexed and ALL respectively.

**int32_t getVal( instruction &instruc, char allowed ):** This function Fetches the value at the EA(Effective Address) for the operand instruct from RAM. This function will set error flags in the PSR if it is unable use the access mode provided in allowed. The operand 'allowed' should be ORed values from the address mode s in operators.cpp they are identified by the format _AM_XXX and are _AM_DIR , _AM_IMM, _AM_IND, _AM_IDR, _AM_IDI , _AM_ALL, standing for DIRect , IMMediate , INDexed, InDiRect, InDirect Indexed and ALL respectively.\

## OPERATORS:

The Operators are implmented as stated in the **Figure 1.6.** The only important thing to note is that due to abigous specifiations the arithmetic operators on the index registers use the lower 12 bits of its operand.

| Mnem. | Description | Legal Addressing Modes |
|---|---|---|
| HALT | Halt the machine. | Ignored. |
| NOP | Do nothing. | Ignored |
| LD | Load the accumulator from memory. | All. |
| ST | Store the accumulator into memory. | All except Immediate. |
| EM | Exchange the accumulator with memory. | All except Immediate. |
| LDX | Load the specified index register from the upper half of a memory word. | Direct, Immediate. |
| STX | Store the specified index register into the upper half of a memory word. | Direct. |
| EMX | Exchange the specified index register with the upper half of a memory word. | Direct. |
| ADD | Add memory to the accumulator. | All. |
| SUB | Subtract memory from the accumulator. | All. |
| CLR | Clear the accumulator. | Ignored. |
| COM | Complement the accumulator. | Ignored. |
| AND | AND memory to the accumulator. | All. |
| OR | OR memory to the accumulator. | All. |
| XOR | XOR memory to the accumulator. | All. |
| ADDX | Add memory to the specified index register. | Direct, Immediate. |
| SUBX | Subtract memory from the specified index register. | Direct, Immediate. |
| CLRX | Clear the specified index register. | Ignored. |
| J | Jump to the specified memory address. | All except Immediate. |
| JZ | Jump to the memory address if the accumulator contains zero. | All except Immediate. |
| JN | Jump to the memory address if the accumulator contains a negative number. | All except Immediate. |
| JP | Jump to the memory address if the accumulator contains a positive number. | All except Immediate. |

# TESTING:

Several Test files are included with the B17 simulator. Each's output has been hand tested and found to be correct. The Files and expeced output's are as follows:

program1:

```
50 1 000000
c4 5 050404 200800 300800 102840 050c00
101 2 300 9
200 1 30
300 1 10
c4
```

output:

```
0c4:  050404   LD   IMM AC[000050] X0[000] X1[000] X2[000] X3[000]
0c5:  200800   ADD  200 AC[000080] X0[000] X1[000] X2[000] X3[000]
0c6:  300800   ADD  300 AC[000090] X0[000] X1[000] X2[000] X3[000]
0c7:  102840   SUB  102 AC[000087] X0[000] X1[000] X2[000] X3[000]
0c8:  050c00   J    050 AC[000087] X0[000] X1[000] X2[000] X3[000]
050:  000000   HALT     AC[000087] X0[000] X1[000] X2[000] X3[000]
Machine Halted - HALT instruction executed
```

program2:

50 1 000000

100 5 200400 201840 202800 005844 050c84

200 3 30 31 10

100

Output:

100:  200400   LD   200 AC[000030] X0[000] X1[000] X2[000] X3[000]

101:  201840   SUB  201 AC[ffffff] X0[000] X1[000] X2[000] X3[000]

102:  202800   ADD  202 AC[00000f] X0[000] X1[000] X2[000] X3[000]

103:  005844   SUB  IMM AC[00000a] X0[000] X1[000] X2[000] X3[000]

104:  050c84   JN   ??? AC[00000a] X0[000] X1[000] X2[000] X3[000]

Machine Halted - illegal addressing mode

## program3:

50 1 000000

75 3 30 20 10

ff 7 075400 040804 077440 005884 076400 077800 0309c4

ff

## Output:

0ff:  075400   LD   075 AC[000030] X0[000] X1[000] X2[000] X3[000]

100:  040804   ADD  IMM AC[000070] X0[000] X1[000] X2[000] X3[000]

101:  077440   ST   077 AC[000070] X0[000] X1[000] X2[000] X3[000]

102:  005884   CLR  IMM AC[000000] X0[000] X1[000] X2[000] X3[000]

103:  076400   LD   076 AC[000020] X0[000] X1[000] X2[000] X3[000]

104:  077800   ADD  077 AC[000090] X0[000] X1[000] X2[000] X3[000]

105:  0309c4   ???? IMM AC[000090] X0[000] X1[000] X2[000] X3[000]

Machine Halted - undefined opcode

## program4:

```
50 1 000000

c4 5 015400 005804 016440 005884 0a1c40

a1 8 016602 900062 017a02 018682 018400 016a00 1118c1 0b7c80

b7 7 000884 005a41 010a41 000a80 007642 000a82 0e2c00

e2 6 007400 02a804 015984 013904 04b944 050cc0

15 5 10000 20 30 987000 0a1

c4
```

## output:

```
0c4:  015400   LD   015 AC[010000] X0[000] X1[000] X2[000] X3[000]

0c5:  005804   ADD  IMM AC[010005] X0[000] X1[000] X2[000] X3[000]

0c6:  016440   ST   016 AC[010005] X0[000] X1[000] X2[000] X3[000]

0c7:  005884   CLR  IMM AC[000000] X0[000] X1[000] X2[000] X3[000]

0c8:  0a1c40   JZ   0a1 AC[000000] X0[000] X1[000] X2[000] X3[000]

0a1:  016602   LDX  016 AC[000000] X0[000] X1[000] X2[010] X3[000]

0a2:  900062   NOP      AC[000000] X0[000] X1[000] X2[010] X3[000]

0a3:  017a02   ADDX 017 AC[000000] X0[000] X1[000] X2[040] X3[000]

0a4:  018682   EMX  018 AC[000000] X0[000] X1[000] X2[987] X3[000]

0a5:  018400   LD   018 AC[040000] X0[000] X1[000] X2[987] X3[000]

0a6:  016a00   ADDX 016 AC[040000] X0[005] X1[000] X2[987] X3[000]

0a7:  1118c1   COM  016 AC[fbffff] X0[005] X1[000] X2[987] X3[000]

0a8:  0b7c80   JN   0b7 AC[fbffff] X0[005] X1[000] X2[987] X3[000]

0b7:  000884   CLR  IMM AC[000000] X0[005] X1[000] X2[987] X3[000]

0b8:  005a41   SUBX 005 AC[000000] X0[005] X1[000] X2[987] X3[000]

0b9:  010a41   SUBX 010 AC[000000] X0[005] X1[000] X2[987] X3[000]

0ba:  000a80   CLRX 010 AC[000000] X0[000] X1[000] X2[987] X3[000]

0bb:  007642   STX  007 AC[000000] X0[000] X1[000] X2[987] X3[000]

0bc:  000a82   CLRX 007 AC[000000] X0[000] X1[000] X2[000] X3[000]
```

0bd:  0e2c00   J    0e2 AC[000000] X0[000] X1[000] X2[000] X3[000]

0e2:  007400   LD   007 AC[987000] X0[000] X1[000] X2[000] X3[000]

0e3:  02a804   ADD  IMM AC[98702a] X0[000] X1[000] X2[000] X3[000]

0e4:  015984   XOR  IMM AC[98703f] X0[000] X1[000] X2[000] X3[000]

0e5:  013904   AND  IMM AC[000013] X0[000] X1[000] X2[000] X3[000]

0e6:  04b944   OR   IMM AC[00005b] X0[000] X1[000] X2[000] X3[000]

0e7:  050cc0   JP   050 AC[00005b] X0[000] X1[000] X2[000] X3[000]

050:  000000   HALT     AC[00005b] X0[000] X1[000] X2[000] X3[000]

Machine Halted - HALT instruction executed

## program5:

```
50 1 000000

c4 5 015400 005804 016440 005884 0a1c40

a1 8 016602 900062 017a02 018682 018400 016a00 1118c1 0b7c80

b7 8 000884 005a41 010a41 000a80 007642 000a82 007480 0e2c00

e2 6 007400 02a804 015984 013904 04b944 050cc0

5 1 298645

10 1 123456

15 4 10000 20 30 987000

c4
```

## output:

```
0c4:  015400   LD   015 AC[010000] X0[000] X1[000] X2[000] X3[000]

0c5:  005804   ADD  IMM AC[010005] X0[000] X1[000] X2[000] X3[000]

0c6:  016440   ST   016 AC[010005] X0[000] X1[000] X2[000] X3[000]

0c7:  005884   CLR  IMM AC[000000] X0[000] X1[000] X2[000] X3[000]

0c8:  0a1c40   JZ   0a1 AC[000000] X0[000] X1[000] X2[000] X3[000]

0a1:  016602   LDX  016 AC[000000] X0[000] X1[000] X2[010] X3[000]

0a2:  900062   NOP      AC[000000] X0[000] X1[000] X2[010] X3[000]

0a3:  017a02   ADDX 017 AC[000000] X0[000] X1[000] X2[040] X3[000]

0a4:  018682   EMX  018 AC[000000] X0[000] X1[000] X2[987] X3[000]

0a5:  018400   LD   018 AC[040000] X0[000] X1[000] X2[987] X3[000]

0a6:  016a00   ADDX 016 AC[040000] X0[005] X1[000] X2[987] X3[000]

0a7:  1118c1   COM  016 AC[fbffff] X0[005] X1[000] X2[987] X3[000]

0a8:  0b7c80   JN   0b7 AC[fbffff] X0[005] X1[000] X2[987] X3[000]

0b7:  000884   CLR  IMM AC[000000] X0[005] X1[000] X2[987] X3[000]

0b8:  005a41   SUBX 005 AC[000000] X0[005] X1[9bb] X2[987] X3[000]

0b9:  010a41   SUBX 010 AC[000000] X0[005] X1[565] X2[987] X3[000]

0ba:  000a80   CLRX 010 AC[000000] X0[000] X1[565] X2[987] X3[000]
```

0bb:  007642   STX  007 AC[000000] X0[000] X1[565] X2[987] X3[000]

0bc:  000a82   CLRX 007 AC[000000] X0[000] X1[565] X2[000] X3[000]

0bd:  007480   EM   007 AC[987000] X0[000] X1[565] X2[000] X3[000]

0be:  0e2c00   J    0e2 AC[987000] X0[000] X1[565] X2[000] X3[000]

0e2:  007400   LD   007 AC[000000] X0[000] X1[565] X2[000] X3[000]

0e3:  02a804   ADD  IMM AC[00002a] X0[000] X1[565] X2[000] X3[000]

0e4:  015984   XOR  IMM AC[00003f] X0[000] X1[565] X2[000] X3[000]

0e5:  013904   AND  IMM AC[000013] X0[000] X1[565] X2[000] X3[000]

0e6:  04b944   OR   IMM AC[00005b] X0[000] X1[565] X2[000] X3[000]

0e7:  050cc0   JP   050 AC[00005b] X0[000] X1[565] X2[000] X3[000]

050:  000000   HALT     AC[00005b] X0[000] X1[565] X2[000] X3[000]

Machine Halted - HALT instruction executed

## program6:

700 2 0e7058 703e56

700

## output:

700:  0e7058   NOP      AC[000000] X0[000] X1[000] X2[000] X3[000]

701:  703e56   ???? 000 AC[000000] X0[000] X1[000] X2[000] X3[000]

Machine Halted - undefined opcode

## programtest:

50 1 000000

c4 5 015400 005804 016440 005884 0a1c40

a1 8 016602 900062 017a02 018682 018400 016a00

b7 8 000884 005a41 010a41 000a80 007642 000a82 007480 0e2c00

e2 6 007400 02a804 015984 013904 04b944 050cc0

5 1 298645

10 1 123456

15 4 10000 20 30 987000

c4

## output:

ERROR FOUND IN INPUT FILE: INDEX OVERFLOW

Exiting.