COMPUTER
MICRO-PROGRAMMING LEVEL
SIMULATOR
GRADUATE PROJECT REPORT

by FANG HU

COMMITTEE MEMBER

Dr. PATRICK R. MICHAUD, Chairman

Mr. DAVID C. PERKINS, Member

Dr. DAVID E. LEASURE, Member

TEXAS A&M UNIVERSITY, CORPUS CHRISTI
February 20, 1996
ABSTRACT

This project is the development of a software simulator that simulates the hypothetical microprocessor described in Andrew S. Tenenbaum's book *Structured Computer Organization*. This simulator can execute programs written in the assembly language for this microprocessor in a subcycle, cycle and machine instruction fashion. The users of this simulator can study the operations of the microprocessor. This simulator can show the contents of the internal registers of the microprocessor and memory locations at discrete times during the simulated execution via the graphical display. This project includes an assembly language translator which can translate the assembly code of this microprocessor to the machine code and check the assembly code's validity.
# Contents

1 Introduction ............................................................................................................. 1  

2 Narrative ................................................................................................................. 3  
   2.1 Exo-Architecture ................................................................................................ 4  
   2.2 Endo-Architecture ............................................................................................ 5  
      2.2.1 The Data Path ............................................................................................ 5  
      2.2.2 Control Portion ......................................................................................... 8  
      2.2.3 Microinstructions ....................................................................................... 8  
      2.2.4 Microinstruction Timing ........................................................................... 10  
      2.2.5 Microinstruction Sequencing ...................................................................... 11  
   2.3 Description of the Simulator ............................................................................. 13  

3 Procedure ............................................................................................................... 15  
   3.1 Overview of Simulator System ......................................................................... 15  
   3.2 Translator Subsystem ....................................................................................... 16  
   3.3 Simulator Subsystem ....................................................................................... 17  
      3.3.1 Operations Subsystem ............................................................................... 19  
      3.3.2 Graphical User Interface Subsystem ........................................................ 27  

4 Results and Conclusions ....................................................................................... 30  

Bibliography .............................................................................................................. 31
1 Introduction

Computer architecture (or computer organization) is a very important subject covering CPU organization, instruction cycles, memory, microprogramming, instruction formats and types, addressing, machine language and so on.

The microprogramming involves the fundamental understanding of how CPU executes a program. This understanding is important in studying operating systems, compiler design, and other computer science subjects.

Using a computer to simulate the operation of the CPU is a better way to assist students in understanding the internal architecture of the CPU. When students operate a simulator to execute their programs step-by-step, they obtain hands-on experience of the internal operation of a computer. Students can see the internal structures of the CPU and communication of the various parts of the CPU via graphical interfaces and animation.

This project COMPUTER MICRO-PROGRAMMING LEVEL SIMULATOR provides a tool to assist students to study the operations of the CPU via vivid dataflow graphics among the parts of the CPU. It combines the structure and behavior of the CPU and overcomes the difficulty that the internal part of the CPU and dataflow among them cannot be seen. It amends the insufficiency of some traditional computer architecture and computer organization textbooks to include translation of assembly language into microcode, addition of new machine language instructions to the machine, and structure charts to show the functionality of the CPU[1][2][3].

This project deals with computer architecture problems which involve many basic computer science concepts: microprogramming, assembly language programming, data struc-
tures, representations of numbers in a computer, instruction cycles, timing, and so on. It also involves computer graphics and event programming methods.
2 Narrative

Modern computers are viewed as architectures which are constructed in a series of levels. In this project, we view our computer as an architecture of two levels --- *exo-architecture* and *endo-architecture*[2].

The exo-architecture is a particular abstraction level as seen by the lowest-level programmers. The primary components of this level include:

- The organization of programmable storage (main memory and registers),
- Data types and data structures, including encoding and representation,
- Instruction formats,
- The instruction set,
- The modes of addressing and accessing data items and instructions, and
- Exception conditions.

The endo-architecture level describes how the physical components realize a particular exo-architecture. The primary components of this level are:

- The capabilities and performance characteristics of its principle functional components,
- The ways in which the components are interconnected,
- The nature of information flow among components, and
- The logic and means by which information flow is controlled.
2.1 Exo-Architecture

The length of each instruction of this machine is 16 bits, and there are three addressing modes: direct, indirect, and indexed.

<table>
<thead>
<tr>
<th>opcode</th>
<th>mnemonic instruction</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000xxxxxxxxxxx</td>
<td>LODD Load direct</td>
<td>ac = m[x]</td>
</tr>
<tr>
<td>0001xxxxxxxxxxx</td>
<td>STOD Store direct</td>
<td>m[x] = ac</td>
</tr>
<tr>
<td>0010xxxxxxxxxxx</td>
<td>ADDD Add direct</td>
<td>ac = ac + m[x]</td>
</tr>
<tr>
<td>0011xxxxxxxxxxx</td>
<td>SUBD Subtract direct</td>
<td>ac = ac - m[x]</td>
</tr>
<tr>
<td>0100xxxxxxxxxxx</td>
<td>JPOS Jump positive</td>
<td>if ac ≥ 0 then pc = x</td>
</tr>
<tr>
<td>0101xxxxxxxxxxx</td>
<td>JZER Jump zero</td>
<td>if ac = 0 then pc = x</td>
</tr>
<tr>
<td>0110xxxxxxxxxxx</td>
<td>JUMP Jump</td>
<td>pc = x</td>
</tr>
<tr>
<td>0111xxxxxxxxxxx</td>
<td>LOCO Load constant</td>
<td>ac = x (0 ≤ x ≤ 4095)</td>
</tr>
<tr>
<td>1000xxxxxxxxxxx</td>
<td>LODL Load local</td>
<td>ac = m[sp + x]</td>
</tr>
<tr>
<td>1001xxxxxxxxxxx</td>
<td>STOL Store local</td>
<td>m[x + sp] = ac</td>
</tr>
<tr>
<td>1010xxxxxxxxxxx</td>
<td>ADDL Add local</td>
<td>ac = ac + m[sp + x]</td>
</tr>
<tr>
<td>1011xxxxxxxxxxx</td>
<td>SUBL Subtract local</td>
<td>ac = ac - m[sp + x]</td>
</tr>
<tr>
<td>1100xxxxxxxxxxx</td>
<td>JNEG Jump negative</td>
<td>if ac &lt; 0 then pc = x</td>
</tr>
<tr>
<td>1101xxxxxxxxxxx</td>
<td>JNZE Jump nonzero</td>
<td>if ac ≠ 0 then pc = x</td>
</tr>
<tr>
<td>1110xxxxxxxxxxx</td>
<td>CALL Call procedure</td>
<td>sp = sp - 1; m[sp] = pc; pc = x</td>
</tr>
<tr>
<td>1111000000000000</td>
<td>PSHI Push indirect</td>
<td>sp = sp - 1; m[sp] = m[ac]</td>
</tr>
<tr>
<td>1111010000000000</td>
<td>POPI Pop indirect</td>
<td>m[ac] = m[sp]; sp = sp + 1</td>
</tr>
<tr>
<td>1111100000000000</td>
<td>PUSH Push onto stack</td>
<td>sp = sp - 1; m[sp] = ac</td>
</tr>
<tr>
<td>1111110000000000</td>
<td>POP Pop from stack</td>
<td>ac = m[sp]; sp = sp + 1</td>
</tr>
<tr>
<td>1111111000000000</td>
<td>RETN Return</td>
<td>pc = m[sp]; sp = sp + 1</td>
</tr>
<tr>
<td>1111111000000000</td>
<td>SWAP Swap ac, sp</td>
<td>temp = ac; ac = sp; sp = temp</td>
</tr>
<tr>
<td>11111110yyyyyyyy</td>
<td>INSP Increment sp</td>
<td>sp = sp + y (0 ≤ y ≤ 255)</td>
</tr>
<tr>
<td>11111110yyyyyyyy</td>
<td>DESP Decrement sp</td>
<td>sp = sp - y (0 ≤ y ≤ 255)</td>
</tr>
</tbody>
</table>

xxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.

yyyyyyyy is an 8-bit constant; in column 4 it is called y.

Figure 1

Instructions using the direct addressing mode contain an absolute memory address in
the low-order 12 bits. Such instructions are useful for accessing global variables. The indirect addressing mode allows the programmer to compute a memory address, put it in the register AC (Accumulator), and then read or write the word addressed by PC. This mode of addressing is very general and is used for accessing array elements. The index addressing mode specifies an offset from SP (Stack Pointer) and is used to access local variables.

The instruction set and instruction format of this machine is shown in Figure 1[1]. Each instruction contains an opcode and sometimes a memory address or a constant.

In this project, the memory size is 4096 word (the length of a word is 16 bits), the range of address is from 0 to 4095. This machine can perform addition or subtraction on the integer data type. Other operations (multiplication, division and so on) are implemented in software at the exo-architectural level.

2.2 Endo-Architecture

There are two portions of the CPU: the data path portion and the control store portion.

2.2.1 The Data Path

The data path is the part of the CPU containing the ALU, its inputs, and its outputs. It contains sixteen identical 16-bit registers, labeled:

- **PC** Programming Counter.
- **AC** Accumulator.
- **SP** Stack Pointer.
IR Instruction Register, holds the machine instruction currently being executed.

TIR Temporary copy of IR, used for decoding the opcode.

0 Holds zero.

+1 Holds positive one.

-1 Holds negative one.

AMASK Address mask, 007777 (octal), is used to separate out opcode and address bits.

SMASK Stack mask, 000377 (octal), is used in the machine language instructions for such actions as increasing or decreasing the stack pointer to isolate the 8-bit offset.

A, B, C, D, E, F Have no pre-arranged functions and can be used as the microprogrammer wishes.

These registers form a scratchpad memory accessible only at the microprogramming level. Each register can output its contents onto one or both of two internal buses, the A bus and the B bus, and each can be loaded from the third internal bus, C bus (Figure 2).

The A and B buses feed into a 16-bit-wide ALU that can perform four functions:

- plus \((A + B)\),
- logical AND \((A \text{ AND } B)\),
- pass through \(A \text{ (} A\text{)}\),
- negate \(A \text{ (Not } A\)).
The ALU generates two *status bits*, \( X \) and \( Z \), based on the current ALU output: \( X \) is set when the ALU output is negative, and \( Z \) is set when the ALU output is zero.

The ALU output goes into a *shifter*, which can shift the contents by one bit in either direction, or not at all. A bus and B bus feed latches *A-latch* and *B-latch* respectively because the ALU is a combinational circuit — it continuously computes the output for the current input and function code. This can prevent erroneous operation of ALU from the changes on the buses as a new value is being stored in the scratchpad.
To communicate with memory, a *Memory Address Register* (MAR) and a *Memory Buffer Register* (MBR) are included in the micro-architecture. The MAR can be loaded from the B-latch in parallel with an ALU operation. On writes, the MBR can be loaded with the shifter output in parallel with, or instead of, a store back into the scratchpad. On reads, the data read from memory can be presented to the left input of the ALU via the AMUX (*A multiplexer*).

### 2.2.2 Control Portion

The largest and most important item in the control portion of the machine is the control store. This is a special high-speed memory where the microinstructions that define each of the machine instructions are kept. In our control store microinstructions will be 32 bits wide and the microinstruction address space will consist of 256 words.

### 2.2.3 Microinstructions

The functions of the microinstruction will consist of gating values onto the A and B buses, latching them in the two bus latches, running the values through the ALU and shifter, and finally storing the results in the scratchpad and/or MBR. In addition, the MAR can also be loaded, and a memory cycle initiated. To control the data path we need 22 signals which listed as follows:

- 1 bit for controlling of AMUX,
- 2 bits for controlling of ALU operations,
- 2 bits for the shifter.
• 1 bit each for controlling MBR, MAR, READ, WRITE, and ENC which control the storing of data into the scratchpad register, and

• 4 bits each for A, B, C buses indicating which scratchpad register the data on the bus is from.

In order to control jumping and transferring of the microprogram, we add two additional fields COND (2 bits) and ADDR (8 bits). In total, a microinstruction has 13 fields and a length of 32 bits (Figure 3). The detailed functions of each field are as follows:

<table>
<thead>
<tr>
<th>AC</th>
<th>COND</th>
<th>ALU</th>
<th>SH</th>
<th>MAR</th>
<th>RD</th>
<th>WR</th>
<th>ENC</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>ADDR</th>
</tr>
</thead>
</table>

**Figure 3**

**AMUX** control left ALU input: 0 = A-latch, 1 = MBR.

**COND** 0 = no jump, 1 = jump if N = 1, 2 = jump if Z = 1, 3 = jump always.

**ALU** ALU function: 0 = A + B, 1 = A AND B, 2 = A, 3 = NOT A.

**SH** shifter function: 0 = no shift, 1 = right 1 bit, 2 = left 1 bit, 3 = not used.

**MBR** loads MBR from shifter: 0 = don’t load MBR, 1 = load MBR.

**MAR** load MAR from B-latch: 0 = don’t load MAR, 1 = load MAR.

**RD** requests memory read: 0 = no read, 1 = load MBR from memory.

**WR** requests memory write: 0 = no write, 1 = write MBR to memory.

**ENC** control storing into scratchpad: 0 = don’t store, 1 = store.

C select register for storing if ENC = 1: 0 = PC, 1 = AC, ..., 15 = F.
B select B bus source: 0 = PC, 1 = AC, .... 15 = F.

A select A bus source: 0 = PC, 1 = AC, .... 15 = F.

Note that the field order in MIR is arbitrary. In order to minimize line crossings on the graph, the field order is arranged as shown in Figure 3.

We write the microprogram in a Pascal-like procedural language, and translate this microprogram into binary code with width of 32 bits. Figure 4 is the table of example microinstruction statements and corresponding binary code\(^1\). All micro-instructions are combination of these codes.

<table>
<thead>
<tr>
<th>A C</th>
<th>M O A</th>
<th>M M E</th>
<th>A D</th>
</tr>
</thead>
<tbody>
<tr>
<td>M O</td>
<td>U X A</td>
<td>N L S</td>
<td>H R D</td>
</tr>
</tbody>
</table>

- mov := pc; rd
- rd
- ac := mbr
- pc := pc + 1
- mov := ir; mbr := ac; wr
- ac := in; if n then goto 15
- ac := nrtr
- ir := ishift(ir); if n then goto 25
- ac := ac; if z then goto 22
- ac := ac; if z then goto 69
- sp := sp + (c-1); rd
- ir := ishift(a + ir); if n then goto 69

Figure 4

2.2.4 Microinstruction Timing

A basic ALU cycle consists of initializing the A and B latches, giving the ALU and shifter enough time to do their work, and storing the results into the scratchpad. Each
microinstruction can be finished in one clock cycle except read from and write to memory. In order to achieve the correct event sequencing, we divide one clock cycle into *four subcycles* which are controlled by a four-phase clock. The key events during each of the four subcycles are as follows:

**subcycle 1** Load the next microinstruction to be executed into a register called the MIR, the *MicroInstruction Register*.

**subcycle 2** Gate scratchpad registers onto the A and B buses and capture them in the A and B latches.

**subcycle 3** When the inputs are stable, give the ALU and shifter enough time to produce a stable output and load the MAR if required.

**subcycle 4** When the shifter output is stable, store the C bus contents into the appropriate scratchpad register and load the MBR, if either is required.

**2.2.5 Microinstruction Sequencing**

Like any other memory, the control store needs a register to index into control memory and a register to buffer the values read from the control memory. We call the indexing register the MPC (*Microprogram Counter*) because its only function is to point to the next microinstruction to be executed. The buffer register is just the MIR as mentioned above.

Some of the time it is sufficient just to fetch the next microinstruction in physical sequence, but we also need to allow conditional jumps in the microprogram in order to enable it to make decisions. For this reason we provide two fields in each microinstruction: ADDR, which is the address of a potential successor to the current microinstruction,
and COND, which determine whether the next microinstruction is fetched from MPC + 1 or ADDR.

The choice of the next microinstruction is determined by the Micro sequencing logic during subcycle 1, when the ALU output signals N and Z are valid. The output of this component controls the M multiplexer (Mmux), which routes either MPC + 1 or ADDR to MPC, where it will direct the fetching of the next microinstruction.

Because the main memory access time is longer than a clock cycle, when we assert the
memory RD line in one clock cycle, we cannot get the data from the memory until the next clock cycle. Assuming each main memory access takes two clock cycles, we also need to assert the RD line the next microinstruction executed. In the same way, a memory write takes two microinstruction cycles to complete. In our simulation, we need to assert the RD or WR line for two consecutive clock cycles whenever we need to read data from the main memory or store data into the main memory.

2.3 Description of the Simulator

This project simulates the machine with the above characteristics. The result is a graphical user interface (Figure 5) such that users can operate this machine to fetch, then execute an machine instruction and finally store the results into memory by pressing the corresponding button icons.

There are eight buttons to operate this machine:

**Update button** to update the display so that it agrees with the internal data structures.

**Off Display button** to turn display off.

**On Display button** to turn display on (the default).

**Memory button** to display the contents of memory in this machine.

**Instruction button** to execute until the start of the next machine instruction.

**Cycle button** to execute until the start of the next microinstruction.

**SubCycle button** to execute until next microinstruction subcycle.

**Quit button** to exit from the simulator.
On the top-right of the simulator display window there are two boxes: Decoded IR and Decoded MIR. Decoded IR box displays the current executing machine instruction in the symbolic form; Decoded MIR box displays the current executing micro-instruction in the symbolic form.

When the user operates this simulator, the data graphically flows from its source to its destination to illustrate the internal dataflow in the CPU, and the corresponding contents of registers are changed.

This project includes a translator which translates user assembly code to the binary code that this simulator describes. Users develop their assembly program with the assembly language described above and use this translator to create programs in binary machine code contained in the memory file (memory). The simulator can then execute this memory file. If user's source assembly program has errors, the translator prints out the error messages to indicate where and what the errors are.