Introduction

As part of their ongoing attempts to reinvent the wheel, only cheaper, Robertics, Inc. has announced a new microprocessor line to compete for future state contracts. The first product that includes this processor, the WFBCSC-HSE, has an “official” introduction date of July 4, 2004, but should start appearing in the stores some weeks earlier.

The product, the wing-fryer beer-cooler stereo-controller (homeland security enhanced), will be controlled by the Robertics, Inc, Miracle processor core, and will include all the functionality of the Robertics WFBCSC I, the original wing fryer-beer cooler-stereo control system. Many high state officials were WFBCSC owners when they were in college, and remember (or think they remember) them fondly. Indeed, Governor Rowland reportedly was very fond of his, which he received as a gift from one of his friends (“I’ve been fortunate in having generous friends” reported the Governor.) Although, it is said, you cannot fool all of the people all the time, you can in fact fool some of the people all of the time, so this product has a good chance of success.

“Besides, this model is homeland security enhanced” said a company spokesman. “We cannot tell you exactly what it does due to security considerations, but suffice it to say that any true American would respond to a Code Red by eating hotter wings, drinking colder beer, and playing louder music.”

Except for the processor, most of the hardware is the same as the WFBCSC I. Missing from the press release, however, was mention of the Robertics CEO’s current compensation package, that has essentially siphoned money from all ongoing projects, leaving $27.35 as the remaining development budget. To avoid financial disaster, all at the company agree: it’s time to bring in the unpaid student labor.

Fortunately, the processor hardware has been completed, but without the control that allows it to run programs. Your job is to provide the control circuitry that specifies how the processor implements its commands and programs.

The hardware

The processor architecture is shown in Figure 1. It is a von Neumann architecture, using the same memory for program steps and data. There is a single 8-bit bus, which is used to move data from component to component. The memory is 256 bytes, accessible using two registers, one for address, one for data, both connected to the bus. The program counter (PC) is an 8 bit register that is used to keep track where the current (and next) instruction is in memory. The instruction register (IR) is used to store an instruction, which is loaded from memory using the bus. There is a bank of 16 registers R0 - R15 that are used for holding data for manipulation; only one at a time can be connected to the bus for reading or writing. Arithmetic and other functions can be done using an 8-bit ALU; its inputs are an 8-bit register (AIR) for the B inputs, and the bus for the A inputs. The output of the ALU is clocked into another register (the AOR); from there the output can be routed to the bus.

Memory design

The microprocessor must interface with a memory unit, which is of course already designed for you. The memory unit has:

1. an 8 bit data bus which acts an input to the memory during writes and acts as an output from the memory during reads. This is connected to the MDR of the microprocessor.

2. an 8 bit address input, which means the memory holds 256 different bytes of data. This is connected to the MAR of the microprocessor.

1Named to correspond to the slogan used in Robertics advertisements: “If it works, it’s a Miracle.”
3. A one bit Read input (/R) and a one bit Write input (/W) which control respectively whether the memory operation is a read from memory (/R active) or write to memory (/W active). If neither /R nor /W is active, no action is taken, and the data bus is automatically tristated to a high impedance state.

In general, to write 8 bits of data to the memory, the following steps are executed by the microprocessor:

1. Load the MAR with the address to write to.
2. Load the MDR with the data to write.
3. Assert /W.

Once /W is asserted, the data in MDR are written to the memory. This is asynchronous to match the memory, so care needs to be taken that the value is in the MDR before /W is asserted (which means that writing is done after the MDR value is clocked in).

To read 8 bits of data from the memory, the following steps are executed:

1. Load the MAR with the address to read from.
2. Assert /R.
Once \( /R \) is asserted, the data are read from the memory into the MDR on the next rising clock edge.

NOTE: If both \( /R \) and \( /W \) are asserted, the memory acts as if only \( /R \) is active and performs a Read operation on the next rising clock edge.

The memory will be used to store both data and instructions.

The instructions

Instructions are either 8 or 16 bits long (depending on the instruction). For all of these, the first 4 bits (high order) contain the code for the operation. Most instructions involve use of a register \( R0 - R15 \), which is specified as a 4-bit number in the next 4 bits. If an instruction refers to memory (or a number, in the case of LOADI), the location is given in the second 8 bits.

The supported instructions are:

- **CLRR Ra** clear register: stores 0 in register Ra
- **ADDR Ra** add registers: add the number in Ra to Rb, store the result in Rb (Note: \( b = a + 1 \) if \( a \) is even, \( b = a - 1 \) if \( a \) is odd).
- **SUBR Ra** subtract registers: Subtract Rb from Ra, store result in Ra. (a and b as with ADDR).
- **INCR Ra** increment register: adds 1 to register Ra, where Ra is one of \( R0 - R15 \)
- **DECR Ra** decrement register: subtracts 1 from register Ra, where Ra is one of \( R0 - R15 \)
- **BR Ra** Branch to the address stored in Ra
- **BZR Ra** Branch to the address stored in Ra if the zero condition code is true.
- **LOADI Ra, I** Put the second 8 bits of the instruction into Ra
- **LOAD Ra, M** Put the value at memory location M into Ra
- **ADD Ra, M** Add the number in register Ra to the number at memory location M, put the result into Ra
- **SUB Ra, M** Subtract the number at memory location M from the number in Ra, result into Ra
- **STORE Ra, M** Put the value in Ra into memory at location M
- **B M** Branch to location M.
- **BZ M** Branch to location M if the zero condition code holds.

These are described in more detail later.

How it works

An instruction is executed by performing the following sequence of operations:

1. Fetch the instruction to be executed from memory.
2. Decode and execute the instruction.
3. Go to 1.

These operations are all executed by copying data from one microprocessor component to another. We look at these in more detail.
Instruction Fetch

(Note: In the following project description, the notation 'A ⇒ B' means ‘copy the value A to storage location B.’)

The following steps are performed during an 8 bit instruction fetch:

1. PC ⇒ MAR (Prepare to get the current instruction from memory)
2. PC + 1 ⇒ AOR (calculate the address of the next instruction)
3. AOR ⇒ PC (address of the next instruction into the PC)
4. Memory ⇒ MDR (get the instruction)
5. MDR ⇒ IR (put the instruction in the IR where it will be decoded)

For example, if the next instruction to execute is at address 31, then the PC contains the value 31. The fetch steps then proceed as follows.

1. PC ⇒ MAR (Put '31' into the MAR)
2. PC + 1 ⇒ AOR (set the AOR to 32)
3. AOR ⇒ PC (PC gets 32)
4. Memory ⇒ MDR (put the instruction at address 31 into MDR)
5. MDR ⇒ IR (put the instruction (now in the MDR) into IR for decoding)

The following steps are performed during each 16 bit (2 byte) instruction fetch:

1. PC ⇒ MAR (Prepare to get the current instruction from memory)
2. PC + 1 ⇒ AOR (calculate the address of the next instruction)
3. AOR ⇒ PC (address of the next instruction into the PC)
4. Memory ⇒ MDR (get the instruction)
5. MDR ⇒ IR (put the instruction in the IR where it will be decoded)

Upon decoding, we realize it's a 16 bit instruction, and we have to get the next 8 bits, so...

6. PC ⇒ MAR (Prepare to get the next 8 bits from memory)
7. PC + 1 ⇒ AOR (address of the next byte of the instruction)
8. AOR ⇒ PC (set the PC for the next byte)
9. Memory ⇒ MDR (get the rest of the instruction)

If these are done on successive clocks, the process for fetching an 8 bit instruction takes five clock cycles, while fetching a 16 bit instruction takes nine if we leave the second byte in MDR. Note that steps (4) and (9), transferring something from memory, was explained above under the MEMORY section. If we are just a little bit clever we can reduce these to 3 and 5 cycles respectively by doing some work in parallel.
Instruction decoding and execution

Once the instruction is fetched from memory and placed in the IR, it must be decoded and executed. The following is the format of an instruction. For details refer to table below.

First 4 bits of instruction: OPCODE (This tells you what instruction it is). Next 4 bits, the ID of the register used. If it is a 16 bit instruction, the next 8 bits are a memory location (or number in the case of LDI).

The 4-bit op codes, instruction length, and an example command and its code is in the following table:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>OPCODE</th>
<th>length</th>
<th>Example</th>
<th>Ex. instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRR</td>
<td>0001</td>
<td>8</td>
<td>CLRR R15</td>
<td>0001 1111</td>
</tr>
<tr>
<td>ADDR</td>
<td>0010</td>
<td>8</td>
<td>ADDR R8</td>
<td>0010 1000</td>
</tr>
<tr>
<td>SUBR</td>
<td>0011</td>
<td>8</td>
<td>SUBR R4</td>
<td>0011 0100</td>
</tr>
<tr>
<td>INCR</td>
<td>0100</td>
<td>8</td>
<td>INCR R5</td>
<td>0100 0101</td>
</tr>
<tr>
<td>DECR</td>
<td>0101</td>
<td>8</td>
<td>DECR R14</td>
<td>0101 1110</td>
</tr>
<tr>
<td>BR</td>
<td>0110</td>
<td>8</td>
<td>BR R12</td>
<td>0110 1100</td>
</tr>
<tr>
<td>BRZ</td>
<td>0111</td>
<td>8</td>
<td>BRZ R3</td>
<td>0111 0011</td>
</tr>
<tr>
<td>LOAD</td>
<td>1000</td>
<td>16</td>
<td>LOAD R4, 27</td>
<td>1000 0100 00011011</td>
</tr>
<tr>
<td>LDI</td>
<td>1001</td>
<td>16</td>
<td>LDI R3,7</td>
<td>1001 0011 00000111</td>
</tr>
<tr>
<td>ADD</td>
<td>1010</td>
<td>16</td>
<td>ADD R12, 33</td>
<td>1010 1100 00100001</td>
</tr>
<tr>
<td>SUB</td>
<td>1011</td>
<td>16</td>
<td>SUB R1, 128</td>
<td>1011 0001 10000000</td>
</tr>
<tr>
<td>STORE</td>
<td>1100</td>
<td>16</td>
<td>STORE R2, 24</td>
<td>1100 0010 00011000</td>
</tr>
<tr>
<td>B</td>
<td>1110</td>
<td>16</td>
<td>B 17</td>
<td>1110 xxxx 00010001</td>
</tr>
<tr>
<td>BZ</td>
<td>1111</td>
<td>16</td>
<td>BZ 18</td>
<td>1111 xxxx 00010010</td>
</tr>
</tbody>
</table>

The 16 bit instructions are a little different. They require an extra byte after the first to describe a memory location or number. For example, the instruction LOAD R0, 11111111 is encoded as 1000 0000 1111 1111: The 1000 is the instruction’s OPCODE, the 0000 specifies register R0, and the 1111 1111 is the memory location whose contents is put into R0.

There are two unused instruction codes. A nice idea would be to design the circuitry to recognize these, have them act as NO-OPs, and turn on an error bit that stays on until reset. Small, but reasonable extra credit.

Example instruction decode and execution

The instruction **INCR R0** (which is 0100 0000) is at memory location 77, and is next to execute. As described above, this is what happens upon fetching the instruction:

1. PC ⇒ MAR
   (The MAR is loaded with 77)
2. PC + 1 ⇒ AOR
   (78 goes into the AOR)
3. AOR ⇒ PC
   (Put 78 into the PC)
4. Memory ⇒ MDR
   (The MAR now contains the instruction 0100 0000)
5. MDR ⇒ IR
   (The IR now contains the instruction 0100 0000)
6. Now, to decode the instruction, start with the opcode. Since it is 0100, we know we must do an increment operation, so set the ALU control for “A plus 1”

Next, execute the following steps:
(a) R0 ⇒ AIR
(b) AOR ⇒ R0

These particular steps assume we do increment using the A plus 1 function on the ALU. There are other possibilities.

7. Go back and fetch next instruction (go to step 1)

The thing to realize is that each step involves transferring data from one component to another. How do you do this? Since we are sharing a bus, if we want to write from component A to component B, set the appropriate control signals that connect A’s outputs and B’s inputs to the bus, as well as enabling A’s output and B’s load. Then the data is transferred on the next clock.

Each step corresponds to a state, where the control outputs generated for each state are a function of the state and the values of the various operands. Thus you can make a state transition diagram which implements any sequence of copy operations.

The zero condition code

The zero condition code (ZERO-CC) is a logical value used in conditional branching: for BRZ and BZ, you branch to the appropriate address if this code is asserted, otherwise you continue with the next instruction in memory (the address in the PC). NOTE: The zero-CC output of the AOR does not work correctly for this purpose; it is written whenever the AOR is loaded from the ALU, which is on every clock, and it is 1 whenever 0000 0000 is the result written to the AOR. You will need to add the appropriate hardware so that you update this code only when appropriate—whenever a result that you use comes from the ALU, and at most once per instruction. You will need to add the appropriate control logic to implement this. The appropriate time to do this is when you move the AOR result to a register or to the MDR if writing memory; you should not modify it during instruction fetch.

Generating control signals

Each instruction can be implemented as a sequence of data transfers; view this as a sequence of states in a Moore machine, where the state outputs are the control signals to the various components. Where should you start? Here is a basic outline to help you keep the work structured, more details to follow in lecture and discussion.

Once you understand how to copy data from one place to another, draw the state-transition diagram for the 8-bit instruction fetch process, which is simply a sequence of different copy operations (with the exception of reading memory). This diagram should have 4 states, one for each step in the 8-bit instruction fetch process. Next, determine state-transition diagrams for the rest of the instructions. Finally, determine what control values are set for each step of each operation.

Some suggestions:

- Design a method of copying the value in any component in the microprocessor to any other.
- Design Fetch circuitry.
- Design Decode circuitry.
- Design Execution circuitry.

Control signals

Each component has particular control signals (Note—these names are all given as if they were active high, but most are in fact active low. Check the circuit to be sure.):

1. The PC has PC-TO-BUS, which puts the PC contents onto the bus, and BUS-TO-PC, which enables a load from the bus on next clock.
2. The memory signals are BUS-TO-MAR, to load MAR from bus, MDR-TO-BUS and BUS-TO-MDR to put MDR data onto bus and load MDR from bus respectively, also R and W, which causes data to go between MDR and memory as described above. Note that R and W connect to both the memory and MDR modules.

3. For the AIR: BUS-TO-AIR, load AIR from bus; AIR-CLEAR, clear the AIR.

4. For the ALU: the standard control signals as for a 74x181, when considering the inputs as active high (see the description in the 208W online reference book, or a '181 datasheet--this is not the same as is given in Wakerly).

5. For the AOR: AOR-TO-BUS, puts AOR outputs onto bus. Additionally, when AOR-TO-BUS is enabled, zero condition code is set on the next clock (active if AOR = 0000 0000).

6. For the IR: BUS-TO-IR, puts bus value into IR.

7. For the Registers: REG-ADDR, 4 bits to select the register by number; REG-TO-BUS, puts output of specified register onto the bus; BUS-TO-REG, loads specified register from the bus.

8. The ZERO-CC is an output of the AOR, does not go to the bus. Remember that the ZERO-CC output as implemented does not work as is needed for BZ and BZR: you will need to add some more hardware.

These components are fairly well-behaved; any that hook to the bus have 3-state outputs so do not affect bus unless selected. Some care is required with R, W, MDR-TO-BUS, and BUS-TO-MDR.

Details and deliverables

The hardware (in LogicWorks) for the processor will be supplied. The hardware should be fairly free of bugs, and the components have nearly all been designed to display their contents, which should make debugging and testing a lot easier. You will determine the sequence of control signals to implement each instruction, and then document and test your design. There will be more information available as time goes on; we will spend time in both lecture and discussion.

Starting up is to be initiated by asserting an external RESET signal (an input to your control circuit). This should cause the processor to start with the instruction at location 00000000 as soon as this signal is unasserted (on the next clock). Think about what this means in terms of the state machine in your controller and any control signals needed.

Your delivered circuit should include a 'standalone' control circuit whose inputs are the clock, the zero-cc signal, and the contents of the Instruction Register. Its outputs are the set of control signals listed in the Control Signals section above.

Deadlines

The deadlines are extremely tight. No incompletes will be given without a serious reason, and not being done with the project is not one of those. Try to get work done ahead of the schedule and things should work out ok.

13 April Descriptive: You should have decided on all of the discrete steps necessary to implement each instruction. You should list each step, explaining what data is moving where. This should take the form of a rather large state-transition diagram, or a series of diagrams for instruction fetch and individual instructions. This description will allow your TA to help you and provide feedback for the next step. This report is required to be turned in but will not otherwise be graded. To be returned on 4/15.

20 April Functional I. Determine the control values for each step of the fetch cycle, put in a table, and turn it in. These will be discussed in class on 22 April.

22 April Functional II. Determine the control values for the rest of the steps of your instructions. This will be in a chart, and will let you know exactly what each control signal should be during each step of each instruction. Give a detailed explanation of what happens in your
sequence of operations for fetch-decode-execute cycle. Try and make it as detailed (and correct) as possible. Additionally, you will design the state machine that breaks the instructions down to discrete steps. Note: this is due two days after Functional II!

**29 April** Complete. All parts integrated and tested. Your report should contain: a title page with your name, course number, section, and date, objectives, procedure or strategy for solving the problem, functional description of decomposed blocks, diagrams and circuits, test cases and results, discussion, conclusions. Logic works circuits should be accompanied by a disk with the files as well as a hard copy.

All the steps in the design should be shown in the final report.

For this project, the overall strategy and all testing should be explained in detail; there is too much testing to simply print it out! Explain where your circuit works, where it fails, and how it might be improved. Provide detailed instructions so your TA might simulate your the CPU with your controller.

For any help, ask Robert or Gamal. Email is a good way to get in touch.