HOBSys 2009 Machine

Eric Lafferty
UWEC - CS352
laffered@uwec.edu

ABSTRACT

In this paper, I describe how I have mapped the HobSys ISA into a hardware design. This includes a discussion on each piece of hardware used in the design, more specifically it's purpose and reasons for being a part of the design. Diagrams and tables have also been included for quick reference.

1. INTRODUCTION

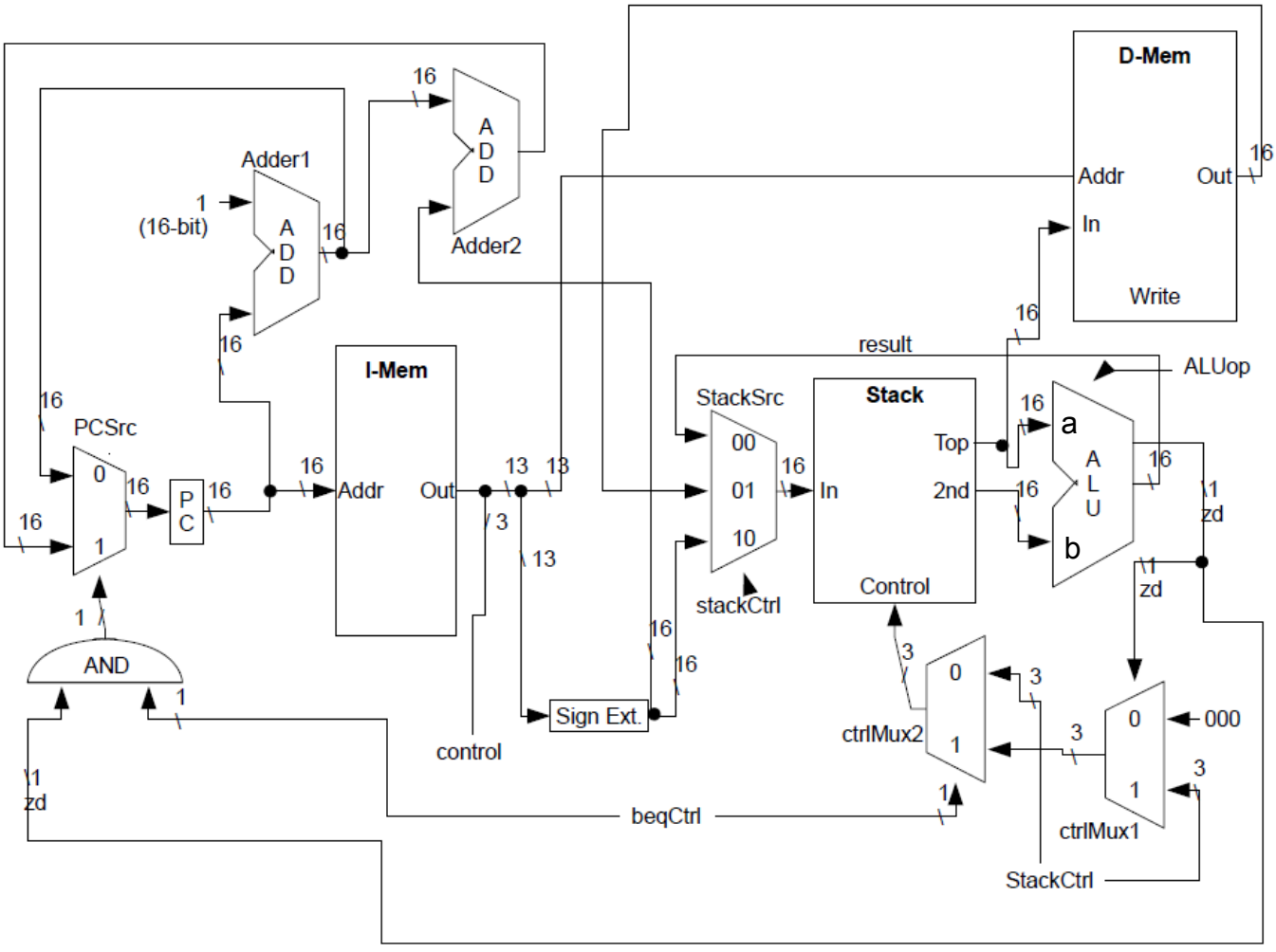
The HobSys machine is a small embedded controller with simple and specific design constraints. It operates on 16-bit values and is a stacked based machine. The two memory structures used are for data memory and instruction memory and both are word addressable, but vary in size. Being a stacked based machine the architecture also implements a stack unit with various control operations. Beyond these three units, an ALU is implemented in the design along with multiple other units including MUXes, ADD gates, and sign extenders.

The ISA operations supported by this architecture include add, nand, push, pop, pushi, beq, nop, and halt. The opcodes for these instructions are 3 bits and are followed in the 16-bit instruction by a 13-bit Z value used as addresses and values.

2. HARDWARE

The hardware implementation of the HobSys ISA can be seen in the diagram below (Figure 1). Each piece of hardware used in the implementation will be discussed in it's own subsection. Included in these will be what the hardware does, what other hardware it directly interacts with, and reasons for why this piece of hardware was selected as part of the implementation.

Figure 1. HobSys 2009 Machine Hardware Design



2.1 Instruction Memory

The instruction memory is word addressable and is capable of holding 128 KBytes of data. This structure has two ports, an address port and an output port. The address port is a 16-bit port and is labeled 'addr'. It is this port which receives it's input from the program counter. The program counter value that is passed to the address port tells the instruction memory which instruction line to load and place on the output port. The output port, labeled 'out', is also a 16-bit port and contains in it the opcode and a Z value. The first three bits of data are the opcode and the last 13 bits are the Z value. The Z value portion of this output is used in two different ways. One way is having it sign extended and then passed to a MUX as a possible input for the stack unit, or to an adder for a branch operation. The other way this value is used is to have it passed into the 'addr' of the data memory. Unlike other memory, the instruction memory is read only and therefore has no write control line.

The instruction memory is a required component of the HobSys machine.

2.2 Data Memory

The data memory is word addressable and is capable of holding 16KBytes of data. The structure has three ports and a write control line. One of the ports is an address port that is labeled 'addr' which specifies the address of the memory to be accessed for reading or writing. This is a 13-bit port which receives its input from the Z value portion of the instruction value passed from the output of the instruction memory. The other two ports are data ports which are labeled 'in' and 'out'. These ports are both 16-bit. All data to be written is directed to the 'in' port, while all data to be read has its source from the 'out' port. The 'in' port receives its 16-bit data from the stack unit port 'Top'. The 'out' port is passed to a MUX as a possible input for the stack unit. The included write control line controls whether a write will be performed to the memory or not. This control line is 1-bit.

The data memory is a required component of the HobSys machine.

2.3 Stack Unit

The stack unit is designed to hold intermediate data much like traditional registers, however done in a fashion similar to the stack data structure. The stack unit has three ports and a control line. One of the ports is for input while two of the ports are used for output. The input port is labeled 'in' and given the proper value in the control line, the value at this port will be pushed onto the top of the stack. The two output ports are labeled 'Top' and '2nd'. The 'Top' port always has the top value from the stack placed on it, regardless of the control signal. The 'Top' passes its value to two locations. One is the ALU port 'a' and the other is the data memory port 'in'. Similarly, the '2nd' port always has the second-from-the-top value placed on it, regardless of the control signal. The '2nd' passes its value to the the ALU port 'b'. The control signal controls what various function will take place on the stack at the end of the clock cycle. This control line is 2-bits.

The stack unit is a required component of the HobSys machine.

2.4 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is a structure that performs various arithmetic and logical operations. The one used in this design has four operations, a corresponding control line, and four ports. The two input ports are labeled 'a' and 'b'. These ports are 16-bits and are used in the different arithmetic and logical operations the ALU performs. Port 'a' receives its value from stack port 'Top' and port 'b' receives its value from stack port '2nd'. The primary output port used is labeled result and is 16-bits. This port provides the result of whatever operation has been performed on 'a' and 'b'. This result is passed to a MUX as a possible input for the stack unit. The other output port used in this ALU is labeled 'zd' or zero detect. This 1-bit output allows for checking of whether or not the operation performed by the ALU resulted in a value that is zero. The value of zero detect is passed to an AND gate and a MUX. The AND gate it is passed to is used to control ,in conjunction with the other MUXes, the way in which to increase the program counter. The MUX gate the zero detect is directly passed to is used in determining what value will be given to the stack control line. More detail on this will be given in the section concerning the 'beq' instruction.

The ALU is a necessary component of the HobSys machine. Without this unit it would take multiple gates in order to do what a single ALU does. Having this piece of hardware simplifies the the operations for add, nand, and beq.

2.5 Program Counter

The program counter unit of hardware starts at zero and keeps track of where the program is in it's execution. It's output is passed to two different places. The first of these is to the instruction memory's 'addr'. This 16-bit value allows the instruction memory to know where in memory the instruction should be taken from. The other place the program counter's output is used is to be added together with the value of 1 by an adder. This is vital so that the machine can continue executing the proceeding lines of code. The input for the program counter comes from a MUX which either passes in the value of PC + 1 or the value of PC + 1 + Z, depending on control values.

This is a necessary part of any machine. Without this piece the sequential execution of instructions would not take place.

2.6 Sign Extend

The sign extend unit of hardware simply extends a 13-bit input to a 16-bit value which it then outputs. The input for this unit is always the value of Z coming from the instruction memory's 'out' port. The output of this unit is used for two purposes. The first is to pass a sign extended version of Z to an adder when used as a program counter offset. This is currently only done with the beq instruction. The other purpose is to pass a sign extended version of Z into a MUX which then allows that value to be an option for the Stack 'in' port. This is required for the pushi instruction.

The sign extend is a necessary component of the HobSys machine. Without this unit the ability to easily sign extend would be eliminated and thus make pushi and beq not functional the way they are defined by the HobSys ISA. While two sign extended values are required in the ISA, by splitting the output of the sign extend I am able to use only a single sign extend.

2.7 ADDers

There are two adders in the proposed hardware design. They both are used for the purpose of generating the appropriate result locations for the program counter to ultimately receive. As discussed previously, the two different program counter inputs that are possible are PC + 1 and PC + 1 + Z. The first adder makes

the PC + 1 option a possibility by simply adding 1 to the current program count and sends it to the PCSrc multiplexer. The second add makes the PC + 1 + Z option a possibility. This is done by adding together the result from the first adder with the sign extend value of Z. The result of this second adder is also passed back to the PCSrc multiplexer. Both of these adder's are identical pieces of hardware. They both have two 16-bit inputs and a single 16-bit output.

The adders were my hardware choice to implement the required features because they do only what is required of them. Another option would be to have ALU's to do the addition operations, but this would be a far more complex piece of hardware than is required for the operation.

2.8 PCSrc (MUX)

The PCSrc multiplexer is placed immediately before the program counter. The output of this multiplexer is passed to the input port of the program counter, both of which are 16-bits. The two inputs from which the multiplexer selects from are the two chooses for increasing the program counter. The first is PC + 1 and the second is PC + 1 + Z. The control line receives it's input from the AND gate which essentially tells this multiplexer if we are branching to a location or just moving forward to the next location. This is a single bit multiplexer.

Because the machine requires the ability to branch, this multiplexer is the most efficient and straightforward way to make this a reality.

2.9 AND Gate

The AND gate takes two input values, applies a logical AND to them and outputs the result. This simple gate was only required once in my implementation of the HobSys machine. One of the inputs it takes in is the zero detect from the ALU and the other input is beqCtrl. The purpose of this gate is for its output to determine a value for the program counter, however the AND gate goes to a MUX which is responsible for the optional values to be passed to the program counter. If the value of the AND gate is 1 this must mean that both the beqCtrl line is set to 1 and the zero detect is set to true. This is what is required because only if these two conditions are true should the program counter be given the value of PC + 1 + Z from the MUX.

2.10 StackSrc (MUX)

The StackSrc multiplexer is placed immediately before the stack unit. The output of this multiplexer is passed to the input port of the stack unit, both of which are 16-bits. This multiplexer has three inputs that come from various controls. The first input is a 16-bit input from the ALU result. The second input is a 16-bit input from the data memory output. The last input is a 16-bit input from the sign extend Z value. This is a two bit multiplexer as it has to be able to handle three different inputs. The control line for this must be set appropriately so that the correct value is passed from the multiplexer to the stack input.

A 2-bit multiplexer is the most efficient way available to be able to select between more than two inputs for the stack. Therefore this is a necessary piece of the hardware design because the ALU result, data memory out, and sign extend Z value are all possible candidates to be the input for stack.

2.11 CtrlMUXs

There are two control multiplexers. As seen on the diagram they are labeled CtrlMux1 and CtrlMux2. Both of them have to do with passing the correct value to the control line of the stack unit.

2.11.1 CtrlMux1

This multiplexer takes in two 3-bit values for input. One of these is the stackCtrl and the other is defined as 000. What controls this multiplexer is the zero detect from the ALU. Given a zero detect of 1 the multiplexer will pass along the stackCtrl to CtrlMux2 because we want to continue the process of executing the stackCtrl. However, this is only the case when we are executing a beq instruction. If we are not then it is necessary to still send along the stackCtrl, but it cannot be dependent on the zero detect. This is where CtrlMux2 comes into play.

2.11.2 CtrlMux2

This multiplexer takes in two 3-bit values for input. One of these is stackCtrl and the other is the resulting output of CtrlMux1. What controls this multiplexer is the beqCtrl and the result of the multiplexer is passed into the control line of the stack. This allows the stackCtrl to always be passed along to the control line of the stack if the beqCtrl is set to 0. In the case that we are executing a beq command, we only want to execute the stackCtrl if the zero detect came out as 1. As previously discussed, exactly this is available and is passed as the second input to this multiplexer from the result of CtrlMux1.

3. INSTRUCTION

The instruction set of the HobSys includes eight different instructions. As stated earlier they are add, nand, push, pop, pushi, beq, nop, and halt. Each of these instructions will be dealt with in specific detail in the following sections. Also, included for reference purposes is a sum of products table (Table 1). This table is relevant for the different controls being set properly depending on the given opcode.

Table 1. Sum of Products

Control	[2]	[1]	[0]
StackSrc =		100	010
BeqCtrl =			101
StackCtrl =	000 + 001	010 + 100 + 101	011 + 101
ALUop =		000 + 101	101
MemWrite =			011

3.1 ADD

The add instruction has an opcode of 000. This instruction pops the top two values off of the stack and pushes their sum back onto the stack. This takes place in hardware using the stack unit and the ALU. The stack provides the two values to be added together from 'Top' and '2nd' and after passing these to the ALU, the ALU has a result with the two of them added together. This is dependent on the ALUop being set correctly to “add” and also the StackSrc multiplexer control line being set to use the result from the ALU and pass that along to the stack 'in' port. The final

control to set is the StackCtrl so that the stack handles what is at the 'in' port properly.

3.2 NAND

The nand instruction has an opcode of 001. This instruction pops the top two values off the stack and pushes the bit-wise NAND of the two values back onto the stack. This operation is very similar to that of ADD. The only difference is with the the ALUop which must be set to "nand." As before with the "add" instruction, the stackSrc multiplexer control line must be set to use the result from the ALU and pass that along to the stack 'in' port. The final control to set is the StackCtrl so that the stack handles what is at the 'in' port properly.

3.3 PUSH

The push instruction has an opcode of 010. This instruction takes the value stored at memory location Z and pushes it onto the stack. This operation requires the stack unit, data memory, and instruction memory. From the output of the instruction memory, the 13-bit Z value is passed to the 'addr' of the data memory. The 'out' of the data memory is than mapped to the StackSrc multiplexer which ultimately goes to the stack 'in' port. This operation also requires that the StackSrc and StackCtrl to be set properly.

3.4 POP

The pop instruction has an opcode of 011. This instruction takes the value at the top of the stack and pops it. It then places that popped value into the memory location specified by Z. This operation requires the stack unit, the instruction memory, and the data memory. It begins with the 13-bit Z value that is retrieved from the 'out' port of the instruction memory. This is passed into the 'addr' port of the data memory. Meanwhile, the stack unit passes the value from 'Top' to the 'in' port of the data memory. As long as the MemWrite control is properly set, the popped data is written to the proper location in memory. This also requires that the StackCtrl is set properly.

3.5 PUSHI

The pushi instruction has an opcode of 100. This instruction pushes the sign-extended version of Z onto the stack. This operation is dependent on the instruction memory, the sign extend, and the stack unit. The 13-bit Z value from the instruction memory 'out' port is passed to the sign extend which is then directed to the StackCtrl multiplexer, which ultimately leads to the stack unit 'in' port. This instruction also requires that the StackCtrl be set properly so that the value is pushed onto the stack.

3.6 BEQ

The beq instruction has an opcode of 101. This instruction checks if the two values at the top of the stack are equal. If they are equal then they are popped off the stack and the program counter is branched to the location PC + 1 + Z. If the two values at the top of the stack are not equal, then nothing happens except a standard increase of the program counter.

This instruction is the sole reason for quite a bit of additional hardware. For one, a second adder was needed in order to add the Z value as an offset to the program counter. Also, I ended up using two more multiplexers (CtrlMuxs) and an AND gate to properly complete the operation. Also need exclusively for this operation is the zero detect output from the ALU.

The beq instruction works by taking the zero detect into a multiplexer and an AND gate. The multiplexer is used in conjunction with another multiplexer in order to properly determine the StackCtrl. The AND gate which uses the zero detect is used to determine whether or not to set the program counter to the PC + 1 + Z value. The beq instruction also uses the sign extend Z which is sent to the second adder. This is used to calculate the PC + 1 + Z value for possible use.

3.7 NOP

The nop instruction has an opcode of 110 and essentially does nothing. It still however increases the program counter by one. As a result this instruction only relies on the the program counter related hardware that increases its value by one. These include the program counter, the first adder, and the PCSrc multiplexer.

3.8 HALT

The halt instruction has an opcode of 111. This instruction increases the program counter and halts the machine. However, the design specifications for this system call for this instruction to not currently be implemented.

4. CONTROL

This implementation of the HobSys hardware uses a five different controls that must vary depending on the opcode. While there are two more controls that are used, one is controlled by the AND gate and the other by the zero detect. The five that are controlled by the controller itself are listed with the appropriate values for each opcode in the following table (Table 2.)

Table 2. Sum of Products

	StackSrc	beqCtrl	StackCtrl	ALUop	MemWrite
add	00	0	100	10	0
nand	00	0	100	00	0
push	01	0	010	x	0
pop	x	0	001	x	1
pushi	10	0	010	x	0
beq	x	1	011	11	0
nop	x	0	000	x	0
halt	x	0	000	x	0

5. MEMORY ISSUES

There is a memory issue that can be the cause of a bit of confusion and thus deserves a bit of discussion. The issue is in regards to how the entire range of memory is addressed in the different memories. The instruction memory holds 128KB and has an addr port size of 16-bits. Furthermore the word size is 16-bit or 2-byte. Given the addr port size, and this word size it is possible to show that the instruction memory is addressable up to the 128KB. This is because 16-bit^2 = 64k words addressable. With each word size being 2-bytes, we can multiple the two numbers together and see it is 128KB.

The issue with the data memory is how to address 16KB of memory with only a 13-bit addr. This again has a word size of 16-bit or 2-byte. Thus being that 13-bit^2 = 8k words addressable, and each word being 2-bytes, we can multiply the two number together and see that the total addressable memory is indeed 16KB.

6. CONCLUSION

The design specified in this document conforms to all the requirements of the HobSys 2009 machine. It implements all the required opcodes and does them clearly and efficiently. The design can clearly be seen in the diagram above (figure 1) and the corresponding information for what each piece of hardware does, what each opcode does, and various explanations where necessary can be found in the appropriately titled sections.