Peppercorn – 16-bit ISA Specification

The purpose of this project was to design a 16-bit ISA from scratch and create the hardware block diagram that would be necessary to implement the CPU in silicon. The hardware block diagram for our architecture, named Peppercorn, exposes the datapath and control logic used to create the CPU and can be found in Figure 7.

This architecture provides a small instruction set that can be used to implement simple, but functional programs. The full list of instructions is ADD, SUB, AND, OR, MOV, MOVI, CMP, B, BEQ, BNE, BR, BREQ, BRNE, LD, ST, and NOP. The architecture is implemented as a load/store architecture with two load/store instructions, LD and ST, respectively. Also included in the instruction set are various ALU operations that can be used to mutate the values of registers. A list of 8 sample instructions and the control signals associated with the instruction can be found in Table 3.

The ISA can be broken up into 5 different classes of instruction encoding. The first kind of instruction encoding is the ALU instruction encoding. The format of an ALU instruction can be seen in Figure 1. An ALU instruction, like all other instructions, begins with a 4-bit opcode field. The opcode field is then followed by three other 4-bit fields, Ra, Rb, and funct. The Ra and Rb fields are 4-bit fields that tell the register file which of the CPU's 16 registers the arithmetic operation should be performed on. Ra and Ra are treated as the source registers for the operation, and Ra is treated as the destination. For example, the RTN for the instruction *ADD R0, R1;* would be *R0* ←*R0* + *R1*. The value in Ra (R0 in this example) is used as one of the addends, and then the result of the addition is placed back in Ra.

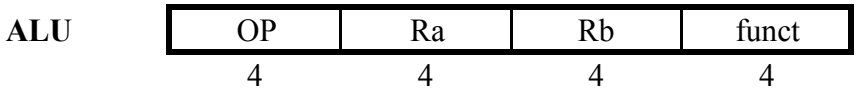

Figure 1: The format of an ALU instruction.

OP	funct[3] (cmp)	funct[2] (mov)	funct[1…0] (ALUop)
ADD	0	0	00
SUB	0	0	01
AND	0	0	10
OR	0	0	11
MOV	0	1	00
CMP	1	0	01

Table 1: The values of the control signals for 8 sample instructions.

Instead of having a control signal for arithmetic logic unit operations, the control signals for the ALU were implemented into the instruction itself. Table 1 demonstrates the various possible ALU

instructions and the corresponding bits that must be encoded into the *funct* field of the instruction to achieve the desired functionality. The upper two bits are reserved to designate if the ALU instruction is a *MOV* or *CMP* instruction, and the bottom two bits indicate the ALU operation to be performed. When a *MOV* instruction is being performed, the ALUop field is set to 0, indicating an ADD operation. As can be seen in Figure 7, when a *MOV* instruction is performed, the ALUINST control signal is high, so the ALU operation is ADD. The upper value of the ALU becomes 0 because ALUINST & ~inst[2] is 1, which results in 0. 0 is then added to the contents of Rb, and stored back into Ra, resulting in the behavior expected from a MOV instruction. The CMP instruction is implemented in the same way, this time with the REGWR control signal low so that the result of the ALU subtraction operation is discarded, leaving only the Z flag changed.

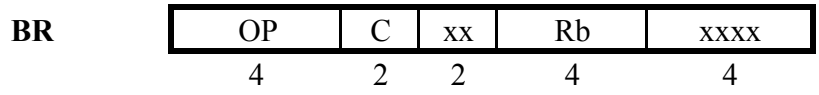

Figure 2: The format of a branch register (BR) instruction.

The next format of instruction is the branch register (BR) instruction. Like the other instructions, the branch register instruction begins with a 4-bit opcode. Following the opcode field is a 2-bit conditional field. This field describes to the branch logic under which circumstances the branch should be performed. Table 2 shows the different possible values for the C field as well as what behavior results from the provided values. For example, if 10b (2) is encoded into the C field, the branch will only be performed when the previous instruction (usually a CMP instruction) did not result in a zero. I.e. the branch will be performed when the two operands provided to the CMP instruction were not equal, hence the mnemonic BNE. Following the 2-bit C field are 2 do-not-care bits. The next 4 bits encode the register number from which the branch address will be loaded. The last 4-bits of the instruction are do-not-care bits.

OP	**C**
UNCOND	00
EQ	01
NE	10
NOP	11

Table 2: The operations for given values of a branch instruction's conditional (C) field.

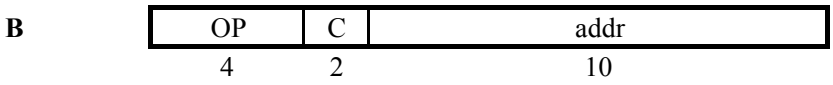

Figure 3: The format of a branch immediate (B) instruction.

For operations such as branching to a label, a branch instruction to which an immediate address can be supplied is optimal. The branch instruction (B) in the Peppercorn architecture will achieve

this functionality. The instruction is encoded starting with a 4-bit opcode, followed by the same 2-bits used to encode the conditional type of the branch that was described in the BR instruction format above. The remaining 10-bits are the address to jump to. The assembler for this architecture will always encode the address of the label provided as relative to the program counter, meaning that if the label is behind the current value of the PC, a negative address will be generated in 2's compliment so that when it is added to the program counter the appropriate address will result and the CPU will be branched to the appropriate location. Both the B and BR instructions can be used to encode the NOP instruction. By setting the C field equal to 11b, a single-cycle no-operation will occur.

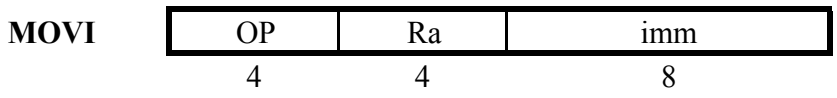

Figure 4: The format of the move immediate (MOVI) instruction.

It is useful to be able to place values into registers without the need to load them in from program memory. The MOVI instruction, of which the instruction encoding is described in Figure 5, enables the architecture to do just this. The MOVI instruction encodes the destination register into which an immediate value should be placed, as well as an 8-bit immediate value. When executed, the 8-bit immediate value will be stored into one of the CPU's 16 registers as held in the Ra field.

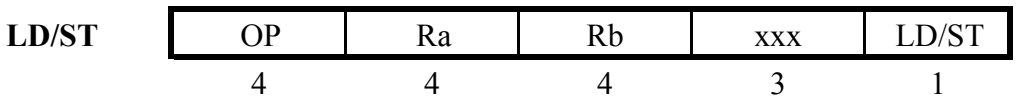

Figure 5: The format of the load/store (LD/ST) instructions.

The most important instruction encoding of this architecture are the load/store instructions. The load/store instructions, LD and ST, respectively, enable the CPU to place 16-bit words from memory into any of the CPU's registers. The encoding is similar to that of the ALU instruction. However, the address held within Rb is used to access memory before storing the value at that memory address back into Rb. The least significant bit of the instruction encodes whether the instruction is a load or store instruction, and is used as a control signal, to save the number of opcodes and control signal lines used. When this bit is 0, the instruction performs a load. When this bit is 1, the instruction performs a store, instead putting the contents of Ra into memory address Rb.

```
/*  This program iterates through an array starting at address R1
 *  with R0 number of elements. The program sets the array element
 *  [word] to 0 if the number in the array is even, and 1 if it is
 *  odd.
 *  R0 - size of the array
 *  R1 - address of array
*/
        movi R2, 0;     // R2 <- 0
        movi R4, 1;     // R4 <- 1
Loop:   cmp R0, R2;     // Check if R0 == R2
        beq Done;       // If so, PC <- &Done
        ld R3, R2;      // R3 <- MEM[R2]
        and R3, R4;     // R3 <- R3 & R4
        st R3, R2;      // MEM[R2] <- R3
        add r2, r4;     // R2 <- R2 + R4
        movi r3, 2;     // R3 <- 2
        add r1, r3;     // R1 <- R1 + R3
        b Loop;          // PC <- &Loop
Done:
```

Figure 6: A sample program that iterates through an array.

As described in the comment, the simple program demonstrated in Figure 6 iterates through an array starting at an address held in R1 with the number of elements stored in R0. It reads the value of each element, and writes back a 0 if the value at that index in the array was even, and a 1 if it was odd. This is accomplished by bitwise ANDing the value loaded from the current index in the array with 1, i.e. taking its least significant bit, and storing it back into the array at the current index. The program also demonstrates the capability of the architecture to implement loops using conditional branching instructions. The mixed capitalization of the register names also demonstrates the capability of the assembler to demonstrate case-insensitivity.

	OP	REGWR	BRANCH	B/BR	ALUINST	LDSTINST	MOVIINST	funct[3]	funct[2]	funct[1]	funct[0]
ADD	0000	1	0	X	1	0	0	0	0	0	0
SUB	0000	1	0	X	1	0	0	0	0	0	1
MOV	0000	1	0	X	1	0	0	0	1	0	0
CMP	0000	1	0	X	1	0	0	1	0	0	1
BR/BRZ/BREQ/BRNE	0010	0	1	1	0	0	0	X	X	X	X
LD	0011	1	0	X	0	1	0	X	X	X	0
ST	0011	0	0	X	0	1	0	X	X	X	1
MOVI	0100	1	0	X	0	0	1	-	-	-	-

Table 3: The values of the control signals for 8 sample instructions.

Table 3 demonstrates 8 different instructions and the values of the control signals needed for the instruction to perform the desired behavior. A – symbol means that an instruction specific bit is placed in that location. A X symbol means that the bit is a do-not-care bit and will be ignored by the control logic for that instruction.

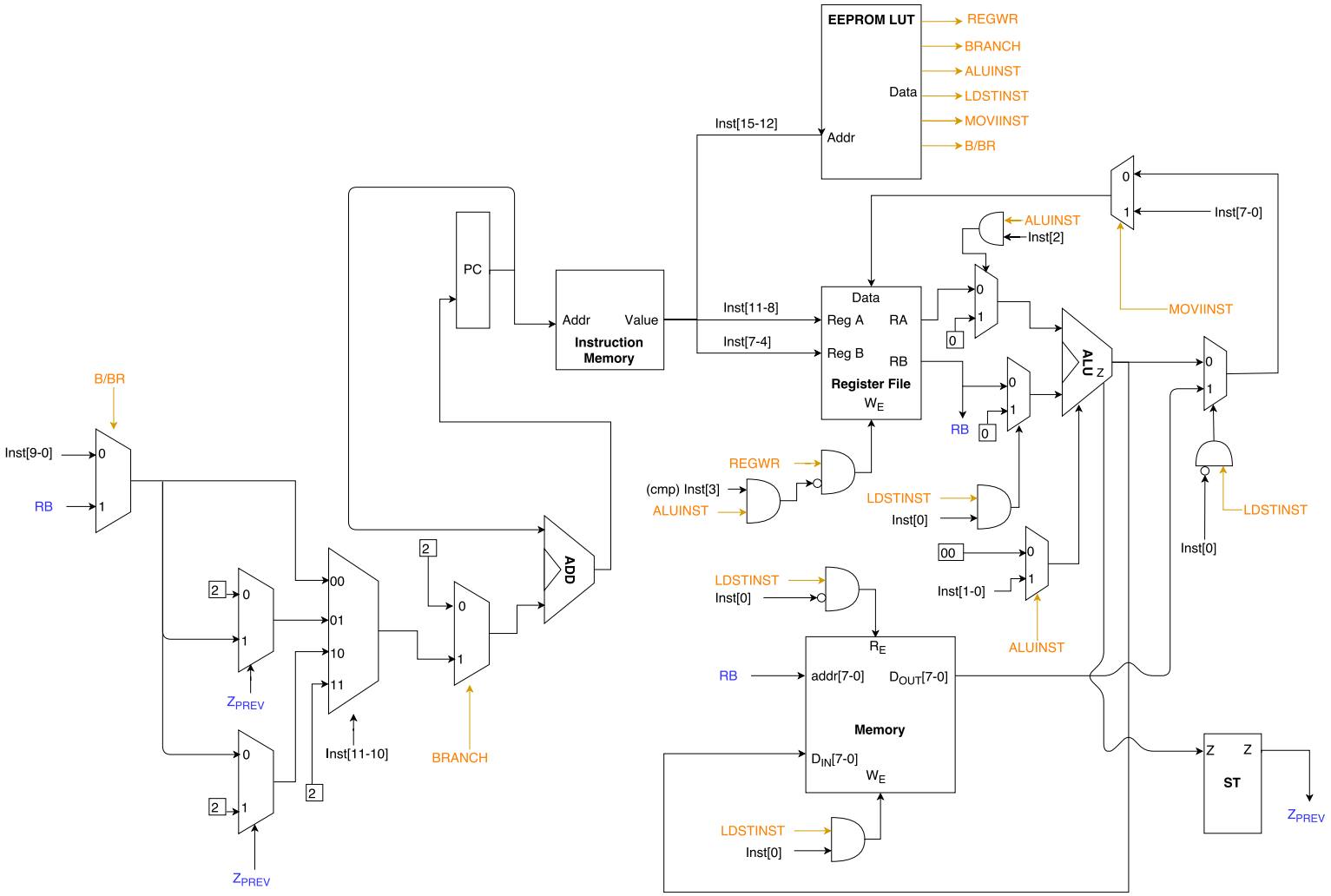

Figure 7: The datapath and control signals for the Peppercorn architecture.

As can be seen in the hardware block diagram, there are two special registers, the PC and ST registers. The PC register is the program counter and stores the current address of execution. The ST register is the status register and contains a single bit flag, the zero flag. The zero flag in the status register contains the zero result of the previous ALU operation. This flag enables the CMP instruction to be used ahead of a branch instruction to conditionally branch. Both the PC and ST registers are updated every clock cycle.

Instruction	Instruction Memory (2ns)	Register Read (1ns)	ALU Operation (2ns)	Data Memory (2ns)	Register Write (1ns)	Delay
	(2ns)	(1ns)	(2ns)	(2ns)	(1ns)	
ALU	2 ns	1 ns	2 ns		1 ns	**6 ns**
Compare	2 ns	1 ns	2 ns			**5 ns**
Branch Imm.	2 ns		2 ns			**4 ns**
Branch Reg.	2 ns	1 ns	2 ns			**5 ns**
Load	2 ns	1 ns	2 ns	2 ns	1 ns	**8 ns**
Store	2 ns	1 ns	2 ns	2 ns		**7 ns**
Move Imm.	2 ns		2 ns		1 ns	**5 ns**

Table 4: The critical path timing table for calculating the required clock period.

Since the critical path, Load, is 8ns long, the clock period of the architecture must be at least 8ns to accommodate for this instruction.