The Simple Computer has instructions that fall in three broad categories. Let’s discuss each category and its instructions. But before that, let’s discuss a concept called *instruction format*.

1. The Instruction Format

The Simple Computer uses 16-bit *instruction words* to specify its operations and operands. Each instruction word specifies one *assembly instruction*. *Assembly language* is the kind of programming that allows a user to specify operations at the processor level. Assembly languages are processor-specific collections of assembly instructions.

The computer’s instruction word is not the same as the datapath’s control word, but they share many similarities. In particular, the bits of the instruction word can be interpreted to determine a datapath control word.

The instruction words of the simple computer are divided into fields. Each field provides a different piece of information about what the corresponding instruction is trying to do. In the Simple Computer, every instruction word is divided into four fields: one seven-bit field, and three three-bit fields. (Different processors employ different sizes and formats for their instruction code words.)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | |  | | |  | | |  | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

The tricky part is that different categories of instructions have different formats for their instruction words – some of the fields mean different things for different instructions. This can make knowing what an instruction word does just by looking at a 16-bit value difficult. But we do have one important fact in our favor: the seven-bit field in each instruction word always represents the same thing.

The seven-bit field contains the instruction’s *opcode*. Each computer instruction has a unique opcode that is contained in its instruction word. This means that if we 1) know the format for every instruction in the processor’s instruction set, 2) determine the opcode of an unknown instruction, and 3) match the opcode of the instruction to that of a specific instruction, then we can determine what the rest of the instruction means.

The format of every instruction is readily available in the table of instructions. As mentioned before, the opcode of every instruction is contained in the first seven bits. Finally, the opcode of every instruction is also contained in the table of instructions. This information allows us to translate back and forth between assembly instructions and instruction code words. For example, this assembly instruction:

ADD R3, R1, R2 // Add the contents of R1 to the contents of R2 and store the result in R3.

corresponds to the instruction word 0000010011001010. The divisions in the instruction word aren’t shown, but conceptually we can divide the instruction word as described above: 0000010 011 001 010. The first seven bits are the opcode, so we can conclude that 0000010 is the opcode for the ADD instruction.

|  |
| --- |
| *Do you have an idea what the other three fields mean for this instruction?* |

(You’ll find the answers to questions like these at the end of the document. Sometimes, the questions will be answered as you read.)

Now that we’ve learned something about how instruction formats work, let’s consider the instruction categories.

1. Register Instructions

The first (and largest) category of Simple Computer instructions is the set of Register instructions.

The Simple Computer is an example of a *load-store architecture*. The variables of a program are generally stored in memory, but the memory values don’t have access to the datapath. This means that if we want to perform operations on variables, we have to *load* the values from memory into registers when we want to use them, and *store* the register results in memory when we are finished with them. (It’s extremely important to recognize that the computer’s *memory* and the computer’s *registers* are different.)

The Register instructions are important because they represent most of the things that we can do with the datapath to perform operations on operands.

The Register instructions share a common instruction word format. Not every instruction in the group uses all of the fields in the format, but all of the fields are available to every instruction in the group. In general, Register instructions operate on the values in registers and place the result into a register.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | | | | | | | Destination Register (DR) | | | Source  Register A (SA) | | | Source  Register B (SB) | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Since the Simple Computer has eight registers in its register set, it should come as no surprise that the register fields of the instruction word are three bits wide.

|  |
| --- |
| *How wide were the register fields in the datapath control word?*  *How do you think the register fields in the instruction code word differ from the register fields in the datapath control word?* |

The ADD instruction is an example of a Register instruction.

ADD R3, R1, R2 // R3 = R1 + R2.

Like most Register instructions, the ADD instruction’s first register is the Destination Register. The second and third register contain the source operands – Source Register A and Source Register B, respectively.

Because of the way our datapath works, not every instruction requires two sources. The Increment (INC) and Decrement (DEC) instructions have *implied operands* of +1 and -1, respectively. Only one register operand is needed.

INC R4, R4 // R4 = R4 + 1 (R4++, for those of you who like to think in terms of C code.)

DEC R2, R2 // R2 = R2 – 1 (R2--)

For both of these instructions, the first register is still the Destination Register. The second register is Source Register A. These instructions don’t have a Source Register B. We leave it out of the assembly instruction entirely.

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the Source Register B field for these instructions?* |

Recall that the Shift Unit in our Simple Computer operates on the B operand only. Thus, Shift instructions appear similar to the Increment and Decrement instructions, but with one important difference:

SL R2, R3 // (Logical) Left-shift R3 and place the result into R2.

SR R4, R4 // (Logical) Right-shift R4 and place the result into R4.

For these instructions the first register is once again the Destination Register, but this time, the second register is Source Register B. These instructions don’t have a Source Register A. As before, we leave it out of the assembly instruction.

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the Source Register A field for these instructions?* |

The Register instruction category also contains the instructions that allow us to access memory. The Load instruction moves a value from memory into a register. In class, we saw that the A operand also serves as the address input to memory. If a register already contains a value, we can use it as a memory address instead of using it as a datapath operand:

LD R4, R1 // R4 = M[R1]

This instruction says “Use the contents of R1 as a *memory address*. Place the contents of that memory location into R4.” In this situation, the first register is the Destination Register. The second register is Source Register A, but here, the register is the “source” of the memory address. These instructions don’t need a second source – the targeted memory location is the “source” of the actual value. So there is no Source Register B in the instruction.

The Store instruction moves a value from a register into memory. The A operand is still able to serve as the address input to memory, but we also saw in class that the B operand also serves as the data input to memory. We can use one register as the address, and another register as the “holder” of the data we want to store.

ST R1, R2 // M[R1] = R2

This instruction says “Use the contents of R1 as a *memory address.* Place the contents of R2 into that memory location.” This time, the first register is Source Register A – the one that holds the address value. The second register is Source Register B – the one that holds the data we want to store. The Store instruction does not have a destination – the targeted memory location is the “destination” of the store. So there is no Destination Register in the instruction.

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the missing field for these instructions?* |

The important thing to remember about Register instructions is that once you know that an instruction is in this category, you can use the format to know which of the instruction word fields need to be filled.

Time-out #1: Processor Control

By now, you should be familiar with the datapath control word. The datapath control word specifies everything that the datapath has to do to carry out its operations.

But the processor is more than just a datapath. At the end of the last section, we described two memory operations. For one of them – the Store instruction – we will have to write a value to memory. We don’t want to write to memory all the time. We just want to do it for that instruction. Therefore, we will need a control bit – a Memory Write input – to determine whether the value of Source Registers B will be written to the memory location specified by the value of Source Register A.

This is a new control bit, so we need it in addition to all of the bits in the datapath control word. Now we have to start thinking about the *processor control word*, which is the datapath control word plus all of the bits that we determine are necessary to perform other instructions.

You might be wondering whether or not we also need a Memory Read bit. We don’t. Recall that the output of the Memory goes into one of the inputs of Multiplexer D. By choosing the right value for the MD bit that is already in the datapath control word, we can choose which value we want to send to the Registers: the value from the Function Unit, or the value from Memory.

We therefore don’t need a Memory Read bit. We just need to choose whether or not we want to route the Memory output (as would happen in a Load instruction) to the Registers.

1. Immediate Instructions

The second category of Simple Computer instructions is the set of Immediate instructions.

Immediate instructions are like Register instructions, in that we want to perform operations on values. The difference is that instead of providing operands from registers, the Immediate instructions provide an operand that is a constant. The value is made available *immediately* (hence the name) instead of needing to be fetched from a register.

The Immediate instructions share a common instruction word format. Not every instruction in the group uses all of the fields in the format, but all of the fields are available to every instruction in the group. In general, Immediate instructions operate on the value in a register and the value of a constant. The result is placed into a register.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | | | | | | | Destination Register (DR) | | | Source  Register A (SA) | | | Immediate Operand (OP) | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

As before, the register fields are three bits wide because there are eight registers. The Immediate Operand field takes the place of the Source Register B field, but it is still only three bits wide.

|  |
| --- |
| *How many different immediate values can we represent using three bits?*  *If the Immediate Operand represents an* ***unsigned value****, what is the range of Immediate Operands that we can choose?* |

The Immediate Operand capability of the Simple Computer is a limitation, but it is one that we have no choice but to work around. Fortunately, we can – we’ll see how when we talk about using these instructions to write programs.

The Add Immediate instruction is an example of an Immediate instruction:

ADI R1, R2, 4 // R1 = R2 + 4

The operand 4 does not represent the value in R4. It literally represents 4 – the number that is one greater than three. Any time you want to add a known constant to the value in some register, and that constant happens not to be one (because if it were, you could use the Increment instruction), the Add Immediate instruction is probably the instruction you want to use.

So, to continue connecting operands to instruction word fields: R1 is the Destination Register, R2 is Source Register A, and 4 is the Immediate Operand.

Here is an example you shouldn’t ever write:

ADI R1, R1, 11 // R1 = R1 + 11 (?)

|  |
| --- |
| *What’s wrong with this instruction?* |

The Load Immediate instruction is very useful for putting constants into registers. If you’ve ever done something like this in C or C++:

int x = 3;

you have used something along the lines of a Load Immediate.

Even though the instruction has “Load” in its name, it doesn’t involve memory. Instead, the constant is being “loaded” directly into the register.

LDI R1, 4 // R1 = 4

In this instruction, R1 is the Destination Register. There is no Source Register A. Instead, the constant value – the Immediate Operand – is the only “source” value involved.

Time-out #2: Processor Control

Fortunately, the Immediate instructions don’t add any control bits to the processor control word. Recall that the datapath control word – which is part of the processor control word – already has a bit that lets us choose whether we want to take Register B as the B operand of the Function Unit, or take a “Constant In” value as the B operand of the Function Unit. That “Constant In” value is our Immediate Operand.

1. Jump and Branch Instructions

None of the instructions we have considered so far allow for conditional execution. Here are some examples of conditional execution that you have probably seen in C and C++;

if(a > 10)

y = x \* 5;

else

y = x \* 2;

for(i = 0; i < 8; i++)

{

sum = sum + data\_array[i];

num++;

}

In the first example, the value of a determines whether we want to change y according to the first instruction or the second one. The instructions are mutually-exclusive, which means that we will end up *doing the first one and skipping the second*, or *skipping the first one and doing the second*.

In the second example, we have a sequence of instructions that we want to do a certain number of times. This means that after incrementing num, *sometimes we want to go back and add the next element of data\_array to sum*. We don’t always want to do this, but “sometimes” – in this case, the first seven times through the loop – we have to do it.

The category of Jump and Branch instructions allow us to move through code in a non-consecutive fashion. For most instructions, once we have done an instruction, we move on to the next one. In general, this allows us to achieve a result by performing one operation at time:

XOR R1, R1, R1 // R1 = 0

ADD R1, R1, R2 // R1 = R1 + R2

SL R1, R1 // R1 = sl(R1)

ADD R1, R1, R3 // R1 = R1 + R3

ADI R1, R1, 5 // R1 = R1 + 5

After making R1’s value 0, we add R2 to R1, which makes R1’s value equal to R2’s value. By shifting R1’s value to the left, we make it equal to R2’s value times two (assuming that overflow doesn’t happen). We then add R3’s value to R1, which makes R1 = 2\*R2 + R3. Adding 5 makes R1’s final value equal to 2\*R2 + R3 + 5.

Why did we do this? Who knows, but maybe somewhere, there is a block of C or C++ code organized as follows:

int foo(int x, int y)

{

return 2\*x + y + 5;

}

If we make certain that R2 holds the value of x, R3 holds the value of Y, and R1 holds the value of the result, the C function is implemented by the assembly code. *Code like C or C++ does not run native to any processor. To execute high-level language code like C or C++, we must translate it into a format that a processor can understand. You should imagine that the first step in this is translating the C/C++ code into assembly. This is what happens when code* ***compiles****. The second step – translating the assembly code into instruction code words – is called* ***assembly****. The result of compilation and assembly is* ***machine code****, which is just another term for the instruction code words that represent the assembly instructions.*

Getting back to the point, we often expect that we want to carry out instructions in order, but as we’ve also seen, sometimes we don’t. We therefore need to have the means to “move around in code” – to “jump” from one place in code to another, without doing the instructions that we jump over. The class of Jump and Branch instructions allows us to do this.

Processors have a special register called a *Program Counter*. The Program Counter’s job is to keep track of where we are in the program. Processors like ours have a memory area that is devoted to holding only instruction words. The Program Counter is the Address Register for the Instruction Memory – it “points” to the instruction that the processor is currently performing.

When most instructions finish executing, all we have to do is add one to the Program Counter, so that it “points” to the next instruction in line. But Jump and Branch instructions cause the Program Counter to be loaded with a new value – the Instruction Memory address of the instruction we want to do next. This is how we “skip” instructions – by causing the Program Counter to “point” to a different instruction than the one that just comes next.

The Jump and Branch instructions share a common instruction word format. Not every instruction in the group uses all of the fields in the format, but all of the fields are available to every instruction in the group. Jump and Branch instructions are tricky, because what the Source Register does changes for different instructions

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Opcode | | | | | | | Address Offset (AD) (Left) | | | Source  Register A (SA) | | | Address Offset (AD) (Right) | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

As before, the register field is three bits wide because there are eight registers. The Address Offset is a six-bit signed value that we break into two three-bit pieces. The pieces are placed into the Left (three most significant bis) and Right (three least significant bits) fields. Taken together, the Address Offset represents a *six-bit signed 2’s complement value*.

|  |
| --- |
| *How many different Address Offset values can we represent using six bits?*  *Since the Address Offset represents a* ***signed 2’s complement value****, what is the range of Address Offsets that we can choose?* |

The Jump instruction is one example of a Jump and Branch instruction:

JMP R1 // Load the Program Counter with the contents of R1.

Executing this instruction will cause the program to jump to whatever instruction is in the Instruction Memory location specified by R1’s value. The Jump instruction is an example of an *absolute jump* – the value in the register says exactly where we want to go in the program. It is also *unconditional* – we will always go to that location if the instruction is executed. All of this means that we must *load the new value into the Program Counter*, replacing whatever was there.

As far as instruction word fields are concerned, the register that holds the destination address is Source Register A. The Jump instruction does not use the Address Offset fields.

The Simple Computer has two Branch instructions – Branch on Negative, and Branch on Positive:

BRZ R4, 5 // If the value in R4 is zero, go forward by 5 instructions.

// Otherwise, go to the next instruction.

BRN R3, -6 // If the value in R3 is negative, go backwards by 6 instructions.

// Otherwise, go to the next instruction.

Branch instructions are *relative jumps* – the jump occurs to a location that is measured relative to the current Program Counter value, which is just the location of the Branch instruction. Branch instructions are *conditional* – they will occur if the condition is met, but if the condition is not met, “normal” code execution causes the program to continue with the next instruction. All of this means that that if the Branch condition is met, we must *add the offset value to the Program Counter value and then load this value into the Program Counter*, replacing whatever was there.

As far as instruction word fields are concerned, the register whose value we are using in the comparison is Source Register A. The second operand is the Offset Value. We can determine what goes into the two fields by representing the Offset Value as a 6-bit signed 2’s complement number and then breaking it into two pieces.

* 5 = 000101. Therefore AD (Left) = 000 and AD (Right) = 101.
* -6 = 111010. Therefore AD (Left) = 111 and AD (Right) = 010.

Having Jump and Branch instructions will solve many problems for us. They will allow us to write the equivalent of FOR loops and IF-ELSE structures. But the particular way that these instructions work in the Simple Computer can cause problems for us. When we look closer at assembly code writing using the Simple Computer’s instruction set, we will see how to take advantage of the benefits while overcoming the shortcomings of these instructions.

Time-out #3: Processor Control

The addition of Jump and Branch instructions has a serious impact on the processor control word. Because we might need to place a new value into the Program Counter, we need a control bit to serve as the *Program Counter Load*. If we know that we want to load the Program Counter, we have to know if we are doing a *Jump or Branch*, since those instructions load the Program Counter in different ways. If we are doing a branch, we have to know what the *Branch Condition* is, since we have two such instructions.

Endgame: Putting the Processor Control Word Together

Based on what you’ve now read, we can show what the complete processor control word looks like:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DA | | | AA | | | BA | | | MB | FS | | | | MD | RW | MW | PL | JB | BC |
| 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

The names of the fields have the following meanings:

DA: **D**estination **A**ddress

AA: Operand **A A**ddress

BA: Operand **B A**ddress

MB: **M**ultiplexer **B** Select

FS: **F**unction Unit **S**elect

MD: **M**ultiplexer **D** Select

RW: **R**egister **W**rite Control

MW: **M**emory **W**rite Control

PL: **P**rogram Counter **L**oad Control

JB: **J**ump or **B**ranch Bit

BC: **B**ranch **C**ondition Bit

The top sixteen bits of the twenty-bit processor control word are simply the datapath control word. The lower four bits are the bits that have been revealed in this document. We could have made our instruction words twenty bits wide and simply made the instruction word act as the processor control word. But since we have an interest in compacting information as much as possible, we will instead use the bits of the instruction word to tell us what the bits of the processor control word should be. A *decoding* process will allow us to derive the processor control word from the instruction word.

What’s noteworthy – and relevant for your project – is that we can *reverse the decoding process to figure out what the instruction word should be for an instruction, if we know how the processor has to be controlled for that instruction.*

We’ll pick up the story in class.

Answers to the Questions

1. The Instruction Format

|  |
| --- |
| *Do you have an idea what the other three fields mean for this instruction?* |

The instruction ADD R3, R1, R2 corresponds to the instruction 0000010 011 001 010. Notice that 011 = 3, 001 = 1, and 010 = 2. The values in the fields specify which register is used for that part of the instruction. In this case, the first field is the Destination Register, the second field is the first Source Register, and the third field is the second Source Register.

1. Register Instructions

|  |
| --- |
| *How wide were the register fields in the datapath control word?*  *How do you think the register fields in the instruction code word differ from the register fields in the datapath control word?* |

The register fields in the datapath control word are three-bits wide, just like the register fields in the instruction code word. There is no difference between the two: as we will see, the “easy part” of translating the instruction word into the processor control word is that DA = DR, AA = SA, and BA = SB.

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the Source Register B field for these instructions?* |

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the Source Register A field for these instructions?* |

|  |
| --- |
| *What value do we “have” to put into the instruction word to fill the missing field for these instructions?* |

When an instruction doesn’t use a particular register in its format, that field can be filled with don’t cares.

1. Immediate Instructions

|  |
| --- |
| *How many different immediate values can we represent using three bits?*  *If the Immediate Operand represents an* ***unsigned value****, what is the range of Immediate Operands that we can choose?* |

Three bits can represent eight different anythings. If the Immediate Operand represents unsigned values, then it is most likely that the Immediate Operand can represent values from 0 to 7, inclusive. *This is very limited, but it is the best that we can do off-hand.*

|  |
| --- |
| *What’s wrong with this instruction?* |

11 is not a value that we can represent using three bits, so it is not a valid Immediate Operand for the Simple Computer. We’ll figure out how to “make” constants that are greater than seven soon.

1. Jump and Branch Instructions

|  |
| --- |
| *How many different Address Offset values can we represent using three bits?*  *Since the Address Offset represents a* ***signed 2’s complement value****, what is the range of Address Offsets that we can choose?* |

Six bits can represent sixty-four anythings. If the Address Offset represents *signed 2’s complement values*, then the positive value of largest magnitude is 011111 = +31. The negative value of largest magnitude is 100000 =   
-32. The limits on branching are instructions that are thirty-one lines in front of the branch and thirty-two lines behind the branch.