Frequently Asked Questions for Decoder Project

Question:

May we use helper functions and numerical literals defined as global constants?

Answer:

You may absolutely include helper functions and global constants (define's) as part of your code. The only requirement is that you submit them with your functions. When you submit your code, do not include your main(). I'll insert my own based on the framework presented in the project description.

Question:

When a constant is passed in, how do we distinguish between an 8 bit and a 16 bit constant?

Answer:

By examining the opcode, you can see what register is being used. For example, the opcode 0xb6 =10110110 is a mathematical operation, i.e., the first bit is a 1. The next three bits, 011, indicate it is an addition. The next two bits, 01, indicate the addition is with ACC, an 8-bit register. Therefore, with the operand being a constant, it needs to only be 8 bits, so it is the next byte in memory. If the opcode had been 0xba = 10111010, the nth destination would have been MAR, a 16-bit register. Then the operand would have been a 16-bit constant, i.e., the next 2 bytes in memory.

Question:

What's the difference between MAR being used as a register and MAR being used as an indirect address?

Answer:

A good example of the difference might be the opcode 00001110. This instruction loads MAR with the address from memory currently pointed to by MAR. In other words, MAR = memory[MAR]. Officially, this code is not correct in that it is an 8-bit operation and MAR is a sixteen bit register. See the next question to address that issue.

Question:

How do I handle 16-bit memory operations?

Answer:

A 16-bit operation requires two memory transfers. For example, the opcode 0x09 = 00001001 is a load command that loads ACC (an 8-bit register) with an operand used as a constant. Your decoder code would look like this:

ACC = memory[PC + 1]

The opcode 0x0d = 00001101, however, loads MAR (a 16-bit register) with an operand used as a constant. Your decoder code would look like this:

MAR = (memory[PC+1] << 8) + memory[PC+2]

The "<<8" shifts the most significant byte 8 places to the left leaving zeros in the low byte for the second byte of the operand to be added.

Question:

It is possible to have an opcode with two operands thus creating operand of more than 2 bytes. Do we have to worry about this?

Answer:

No you don't have to worry about situations with operands of more than 2 bytes. It is possible to generate an opcode that requires more than 16-bits of operands. Do not worry about implementing these. The test code will never use an operand of more than two bytes. For example, the opcode 0xbf = 10111111 would be an ADD where both the destination and source are a memory addresses passed as operands.

1st byte	2nd byte	3rd byte	4th byte	5th byte
Opcode	MSB destination	LSB destination	MSB source	LSB source

This of course means that your code will not be a full implementation of the system design.

Question:

I'm having trouble getting started with the project conceptually. Is there anyway I can get a hint as to how to start?

Answer:

Your code should begin by determining which type of opcode we're looking at. You will need to perform bitwise ANDs to separate the bits that identify the opcode, and three or four if-statements should be possible to classify them.

```
// Check if it's a math function
if ((IR & 0x80) == 0x80)
{
}
// Check if this is a memory function
```

```
else if ((IR & 0xf0) == 0)
{
}
// Check branch function
else if ((IR & 0xF8) == 0x10)
{
}
// Otherwise, it's a special opcode or an illegal opcode
else
{
}
```

Within each if condition, you'll need to further break down the opcode. For example, in the branch functions, you'll need to determine which type of branch it is. This can probably be done with a switch-case block:

```
switch (IR & 0x07)
{
case 0: // Unconditional branch -- Load PC with (memory[pc+1] << 8) + memory[pc+2]
case 1: // Branch if ACC=0
case 2: // Branch if ACC!=0
// And so on...
}
```

This project is not so much about difficult algorithms or processing. The difficulty is in picking through the opcodes to make sure you have all of the cases covered.

Question:

What am I supposed to do with the special opcodes?

Answer:

Nothing really. The fetchNextInstruction() function should increment the PC register to the next instruction, but that is it. If the opcode isn't a mathematical operation, a branch operation, or a memory operation, it's a special opcode or an illegal opcode. The first special opcode, 0x18, is a NOP, i.e., no operation. Don't do anything. The second special opcode, 0x19, is the HALT command. The program

framework I gave you is the only thing that uses that, i.e., it is set up to exit the simulator when it receives a HALT command. Otherwise it's an illegal operation. If you want to, you can print an error message, but you don't need to. The code below distinguishes between the different special opcodes, but there's nothing to do.

```
// The following is the logic code for a special opcode or an illegal opcode

{

if(IR == 0x18) // Then it's a NOP -- do nothing

{}

else if(IR == 0x19) // Then it's a HALT -- framework will halt for us

{}

else // Otherwise it's an illegal opcode -- you can print an error message if you want to

{}

}
```