Exercise 1 – Microinstruction Word

Word 1: D – Copy the MDR register to the TOS register

The ALU F0/F1 function bits specify "A or B", but only the "B" input is enabled. This means that the "A" bits are zero - when or'ed with the "B" bits it means the B bits will not be changed. So the effect is that the B input is sent through the ALU to it's output without being changed.

The B input comes from the "0000" B-Bus field in the microinstruction, which is the MDR register.

The ALU output is stored in the register indicated by the "C Bus" field, which is TOS.

So the microinstruction connects MDR to the B bus, sends the value through the ALU unchanged, and the result travels up the C bus to be stored in the TOS register.

Word 2: B – Add one to the PC register

The ALU F0/F1 function bits specify "A + B", but only the "B" input is enabled. This means that the "A" bits are zero - when added with the "B" bits it means the B bits will not be changed. However the "INC" bit is also set, which means that "1" will be added to the B value.

The B input comes from the "0001" B-Bus field in the microinstruction, which is the PC register.

The ALU output is stored in the register indicated by the "C Bus" field, which is PC.

So the microinstruction connects the PC register to the B Bus, the value is incremented by the ALU, and the result travels up the C bus to be stored in the PC register.

Word 3: A – Read from memory at the address equal to the sum of the LC and H registers

The ALU F0/F1 function bits specify "A + B", and both inputs are enabled.

The "A" input comes from the H register since the H output is hard-wired to the ALU's A input.

The "B" input comes from the "0101" B-Bus field in the microinstruction, which is the LV register.

The ALU output is stored in the register indicated in the "C Bus" field, which is the MAR register.

And the memory control unit's "Read" signal is activated.

So the microinstruction adds sends the LV register to the B bus, the ALU adds this to the H register connected to it's A input, the result is stored in the MAR register, and the memory control unit initiates a read from that address.

Exercise 2 – JAM / JMPC

1. **C – 100 hex**

   JAMZ replaces the high-order bit in the 9-bit "Next Address" field with the Z output of the ALU. The "Next-Address" field is 000 (hex) or 0 0000 0000 (binary), and the ALU Z output is "1", so the result is 1 0000 0000 (binary) or 100 (hex).

2. **A – 000 hex**

   JAMN replaces the high-order bit in the 9-bit "Next Address" field with the Z output of the ALU. The "Next-Address" field is 000 (hex) or 0 0000 0000 (binary), and the ALU N output is "0", so the result is 0 0000 0000 (binary) or 000 (hex).

3. **B – 08C hex**

   JMPC ORs the low-order 8 bits of the 9-bit "Next Address" field with the contents of the MBR register. The "Next-Address" field is 000 (hex) or 0 0000 0000 (binary), and the MBR register contains "8C" (hex) or 1000 1100 (binary), so the result is 0 1000 1100 (binary) or 08C (hex).

Exercise 3 – MAL to Microinstruction

A – To send TOS to MDR we have to connect TOS to the B bus, set the ALU to pass the B input through without change, and activate the C Bus signal to load the result into the MDR register.

The "B Bus" field of "0111" connects TOS to the ALU's B input.

The ALU F0/F1 control signals "01" specify "A OR B", and enabling only the B input sends it through the ALU without change.

The C Bus "MDR" bit loads the result into the MDR register.

And the memory control unit "WRITE" signal corresponds to "wr" in the MAL statement.

Exercise 4 – MAL to Microinstruction

B – Note that we don't need to pay any attention to the "goto iload3" portion of the MAL statement since we're not dealing with the "Next address" portion of the microinstruction (and we wouldn't know where the "iload3" microinstruction happens to be without more information anyway).

To add LV and H and put the result into MAR, we need to connect LV to the B Bus, tell the ALU to add A + B (H is connected to the A input), and load MAR from the C bus.

The "B Bus" field of "0101" connects LV to the ALU's B input.

The ALU F0/F1 control signals "11" specify "A + B", and both the A and B inputs are enabled – this corresponds to the ALU "A+B" function. (Note that microinstruction "C" also has the "INC" bit turned on which would result in A+B+1).

The C Bus "MAR" bit loads the result into the MAR register.

And the memory control unit "READ" signal corresponds to "rd" in the MAL statement.

Exercise 5 – MAL to Microinstruction

B – The "if (Z)…" portion of the MAL statement means that the JAMZ bit needs to be on in the microinstruction in order to select one of two possible next addresses based on whether the ALU's Z output is 0 or 1.

Note that the result is being stored in both MAR and SP (MAR = SP = SP – 1).

To subtract one from the SP register and store it in both SP and MAR, we need to connect SP to the B Bus, select the "B-1" ALU function, and set the control signals to load both the MAR and SP registers from the result on the C Bus.

The "B Bus" field is set to "0100" to send the SP register to the ALU "B" input.

To have the ALU perform the "B-1" function, we need to add "B" plus "-1". The ALU F0/F1 signals are set to "11" to select the "A+B" function, but only the "B" input is enabled. This means that the signals coming into the A side of the ALU are all "0"s. To get a "-1" the INVA bit is also set – this flips all of the "0" signals on the A side to "1"s, and the result is that the A side of the ALU is all "1" bits – the signed value for "-1".

Finally, the C Bus field has both the SP and MAR bits set so that the result is loaded into both registers.

Exercise 6 – Main Loop

1. **B – the opcode of the instruction being decoded**

   If we're decoding an IADD instruction, the MBR would be contain hex 60 because that is the opcodes for IADD. If it wasn't hex 60, it would have to be a different instruction.

2. **A – The address of the instruction being decoded**

   The PC (Program Counter) register always points to the instruction being executed.

3. **C – The PC register**

   Since the PC register contains the address of the instruction to be executed, that's the register we use to fetch it.

Exercise 7 – POP Microcode

B – because removing the top word from the stack reveals a different top-of-stack value which may be used in subsequent instructions

The TOS register must always contain a copy of the word at the top of the stack – this means that whenever a the top-of-stack value changes we have to update TOS. The value can change even if we don't put actually write a new value there ourselves – removing the top word from the stack is an example of this.

Exercise 8 – INEG Instruction

A – We have to negate the TOS value and write it to the address in the SP register (the top of the stack). It looks like answer "C" should do this just fine, but the problem is that there's no way for the ALU to give us "-TOS". This is because TOS can only be connected to the ALU's "B" input and there's no "-B" ALU function – only a "-A" function. So in order to negate a number we first need to move it to the H register, where it can then be put into the ALU's A input and negated.

Exercise 9 – The Wide Prefix

E – **116**

The WIDE prefix will OR the opcode of LLOAD (hex 16 or binary 0001 0110) with hex 100 (binary 1 0000 0000) to get the address of the 1st instruction for the wide version of LLOAD. This means that instruction must be at address hex 116 (binary 1 0001 0110).

Exercise 10 – ISTORE_1 Instruction

C – The address of local variable "1" is calculated by adding "1" to the LV register, and the result is placed into MAR so that we can write the top-of-stack value into it. Then we remove the word from the stack by decrementing the stack pointer.

Since we now have a different word on the top of the stack, we need to it into the TOS register. This requires us to wait for a clock cycle while the value read from memory arrives in the MDR register.

This IJVM instruction uses almost the same microinstructions as ILOAD, except that instead of fetching the local variable number from the instruction stream we just use a constant value of "1".