CSC 462/562 Homework #8 answer key

1.
   a. t0: C17 (inc SP)
      t1: C8, C19 (AC → MBR, SP → MAR)
      t2: C1, C3, C5 (MAR → addr bus, memory write, MBR → data bus)
      NOTE: I am assuming the SP is pointing at the current top of stack, so the SP needs to be incremented before we push the new item onto the stack.
   b. t0: C9 (IR → MAR)
      t1: C1, C2, C4 (MAR → addr bus, memory read, data bus → MBR)
      t2: C11 (MBR → MAR)
      t3: C8 (AC → MBR)
      t4: C1, C3, C5 (MAR → addr bus, memory write, MBR → data bus)
   c. t0: C12, C29 (AC → ALU, AC == 0?)
      t1: if true C31 to t2 else C31 to microaddress for microprogram for instruction fetch
      t2: C10, C15, C22, C16 (PC → ALU, IR → ALU, ALU add, ALU → PC)
   d. t0: C9 (IR → MAR)
      t1: C1, C2, C4 (MAR → addr bus, memory read, data bus → MBR)
      t2: C13, C24, C26 (MBR → ALU, ALU negation, ALU → AC)
      t3: C8 (AC → MBR)
      t4: C1, C3, C5 (MAR → addr bus, memory write, MBR → data bus)
   e. t0: C9, C30 (IR → MAR, PC → MBR)
      t1: C1, C3, C5 (MAR → addr bus, memory write, MBR → data bus)
      t2: C10, C25, C16 (IR → ALU, ALU Inc, ALU → PC)

2. All registers and all addresses are 16 bits, which would only permit 64K of addresses (the Intel processor is byte addressable, so this means 64KB). If we just left shifted the address by 4 bits, we would only be able to address every 16th byte (e.g., 0000000000000000 is address 0 and 0000000000000001 is address 1, left shifting gives us 00000000000000000000 which is still address 0 and 00000000000000010000 is 16). So we need to compute the address by left shifting the base by 4 bits and add to it an offset. The base is stored in one of four registers, the CS, DS, ES and SS (code, data, extra and stack segments). The context of generating the address is used to determine which segment register to use: program instructions: CS, data access: DS, stack operation: SS. The ES is used when a portion of memory needs to be used by a different purpose than code, data or stack, or when more memory is needed to support one of these uses. For instance, the extra segment might be used for graphics or to extend the code segment. This segment register is left shifted by 4 bits to form a 20-bit base address. Another register's contents are added to it. This might be the IP (the PC for the Intel processor) which stores the offset to add to the CS (or ES), or the SP (stack pointer) to add to the SS, or an address register storing a pointer like the SI, DI or BP register, which would be added to the DS (data segment).

3.
   a. Started with the 286 via segmentation, enhanced for the 386
   b. 8086: 2 (one for computing an address and one in the ALU), 286: 3 (another adder for computing physical addresses), 386: same 3.
   c. 486 had an 8KB cache. 386, used as a TLB.
   d. 8086: 4 items, 286: 8 bytes, 386: 16 bytes, 486: 32 bytes. The prefetch queue stores machine instructions while the decoded instruction queue stores microcode.

4. One issue is the length of instructions. The 486 fetched 16 bytes but the longest instruction could be 17 bytes. Because of the complexity of instructions, decoding took two pipe stages. This in itself would not cause stalls, but is an added stage that other architectures didn't need. ALU operations would vary in length depending on the type of instruction and whether a datum is coming

from memory. Consider the instruction add a[ebx*4+4], eax. This instruction must first compute ebx*4+4 and then fetch the datum from cache, do the addition, and store the result back to memory. This could take several cycles, stalling this instruction in the pipeline's execute stage. So we see two non-RISC ideas here: complex addressing modes and non-load-store instruction set. Another complex addressing mode which is not allowed in a RISC instruction set is indirect addressing. In RISC, we would accomplish this with two separate instructions such as

```
lw      x2, 0(x1)
lw      x3, 0(x1)
```

In x86, this is handled in a single instruction as in mov eax, @(y). This requires at least two cycles in the EX stage causing a stall. The loop instruction requires decrementing the ecx, comparing the ecx to 0, and if greater than 0, branching. So this instruction requires a decrement, a comparison, and if > 0, an addition (PC + offset). These three steps could probably be done in two cycles but not one. So in this case, we see an instruction that a RISC instruction set would not have because it could not execute in 1 cycle. The branch penalty is 3 cycles because branches are computed in the execute stage (4). This in itself is not too atypical but recall our solution to the RISC-V pipeline. We moved the branch computation mechanism into the second stage. We cannot do that for x86 because of the complex nature of the instruction set. We need the first cycle of decoding to locate the operands that we would then use in the branch instruction. So at best, we could move the branch mechanism into the decode2 stage and have a 2 cycle branch penalty. I don't believe they did this for the 486 so branches remained with a 3 cycle penalty.

5. The deficiencies noted in #4 were variable length instructions, memory ALU operations, complex addressing modes (including indirect), instructions that took multiple cycles to execute, and branches. All microinstructions are the same length and of a single format. So once we have decoded the instruction into microcode, we no longer have a concern over variable number of cycles to fetch and decode instructions. A machine instruction, no matter how complex, is broken into distinct microinstructions, each of which take 1 cycle to execute. Therefore, complex addressing modes wind up being several 1-cycle steps, each of which is its own microinstruction. The same is true of multi-cycle instructions. What microcode does not resolve is branch penalties.

6. The iCore 7 is a multi-issue superscalar using a Tomasulo-style architecture. It contains an instruction fetch unit that includes branch prediction, fetching instructions and filling an instruction queue. The decode unit is unique in that it must convert machine instructions into microcode. The microcode is issued by an issue unit that can issue up to 4 microinstructions at a time. The issue stage uses register renaming via a register alias table and allocator. To handle incorrect branch speculation, the iCore has a reorder buffer (although the buffer stores microinstructions, not machine instructions). The ALU units contain a combined 36 reservation stations to store waiting instructions along with any operands that are already available. The functional units themselves include integer, FP and load/store units. One unit stores addresses so that computed addresses and pointers can be retained. Each core contains its own instruction cache, instruction TLB, data cache and data TLB. Each chip contains a unified L2 cache and a unified L2 TLB.

   Aside from the decode portion, unique to Intel, there is also an arbitrator for each core to deal with clock and power state differences when communicating beyond the core to the L2 caches, other cores, or off the chip itself.