Intel sample problems

1. Explain the role of all 14 of the Intel x86 registers (see slide 2 of the Intel notes).

   Answer:

   IP – instruction pointer (the PC)
   Status flags – store status results of previous computation (e.g., zero, negative, positive, carry, overflow, interrupt, parity)
   AX/EAX – accumulator data register
   BX/EBX – base register (for pointers) and data register
   CX/ECX – counter register (for loop instruction) and data register
   DX/EDX – data register (for input/output) and data register
   SI – source index – used for string movement operations
   DI – destination index – used for string movement operations
   BP – base pointer – used for pointer operations
   SP – stack pointer – used for stack operations
   CS – code segment – stores starting address of block of code, offset by IP
   DS – data segment – stores starting address of block of data, offset by SI, DI, BP, BX
   ES – extra segment – stores starting address of block of extra memory, sometimes used for graphics
   SS – stack segment – stores starting address of block of memory storing stack, offset by SP

2. Notice in slide 2 that the Intel 8086 processor itself contained 40 total pins. Several of these are named AD# as in AD0 and AD14. The "A" stands for address and the "D" stands for data. These pins were used to communicate an address during the first cycle of a bus transfer and used to send or receive a datum in the next three cycles of the bus transfer. Why do you suppose they implemented the chip in this way rather than having separate address and data bus pins? How do you think this impacted the performance of the processor?

   Answer: This concept is known as multiplexing of pins. This was done to reduce the number of pins needed to connect the CPU to the system bus whereby information from the CPU to the system bus could be addresses, data and control signals. To reuse the pins, they had to multiplex or alternate when pins represented an address versus data. So the first bus transfer cycle has the AD pins sending an address and the three remaining bus transfer cycles use the pins for data (either going from CPU to memory or from memory to CPU).

3. For each of the following sets of intel assembly code, provide the pipeline timing diagram. Assume a base-displacement memory reference requires 2 execution cycles, one to compute the address and one to perform the data access, and every memory reference in an ALU operation requires 1 additional clock cycle per memory reference to execute. The number of bytes listed after the instruction is the length of the instruction. Assume an 8 byte-wide data bus to handle instruction and data fetches. All ALU operations take 1 byte to execute except for the mul which takes 4 bytes. Assume forwarding from EX to EX is available. Branches taken accrue a 3 cycle penalty while branches not taken accrue no penalty.

   a.
   ```
   mov     eax, a[ebx*4-4]        (10)
   mul     four                   (4)
   add     x, eax                 (4)
   ```

   b. For the following problem, show the timing diagram assuming the jl is taken.
   ```
   mov     eax, a                 (4)
   ```

```
        mov     ebx, b              (4)
        cmp     eax, ebx            (4)
        jg      foo                 (6)
        jl      bar                 (6)
        mov     a, 0                (8)
        j       next                (8)
foo:    mov     b, 200              (10)
        j       next                (8)
bar:    mov     a, 200              (10)
next:   …
```

c. Repeat b assuming neither jg or jl are taken.

Answers: NOTE: F for fetch, D1 for decode stage 1, D2 for decode stage 2, E for exec, W for write result.

a.

```
        mov     F   F   D1   D2   E    E  W
        mul         s   F    D1   D2   s   E    E    E    E    W
        add                  F    D1   D2   s   s    s    s    E   E   E     W
```

The move takes 2 fetch cycles and 2 execute cycles because of computing the effective address. The multiply is postponed 1 cycle before it can be fetched and stalls before the execute stage starts because of waiting for the memory item to be returned. The add stalls while the multiply executes and then requires 3 cycles of execution because it has to do 2 memory operations, one to read x and one to write x.

b.

```
        mov     eax, a   F   D1  D2   E   W
        mov     ebx, b       F   D1  D2  E   W
        cmp     eax, ebx         F   D1  D2  E   W
        jg      foo                  F   D1  D2  E   (no W)
        jl      bar                      F   D1  D2   E
        mov     a, 0                         F   D1  D2  flush
        j       next                             F   D1  flush
foo:    mov     b, 200                               F   flush
        j       next
bar:    mov     a, 200                                   F   F   D1   D2   E   W
next:   …
```

The jl accrues a 3 cycle branch penalty. The last mov requires 2 fetch cycles. No other penalties/stalls arise.

c.

```
        mov     eax, a   F   D1  D2   E  W
        mov     ebx, b       F   D1  D2  E   W
        cmp     eax, ebx         F   D1  D2  E   W
        jg      foo                  F   D1 D2  E   (no W)
        jl      bar                      F  D1  D2   E   (no W)
        mov     a, 0                        F   D1  D2  E   W
        j       next                            F   D1   D2  E   (no W)
foo:    mov     b, 200                               F    F   D1  flush
        j       next                                      F    F   flush
bar:    mov     a, 200                                         F   flush
next:   …                                                           F   ….
```

Only the first j accrues any penalty (3 cycle branch penalty), no other stalls.

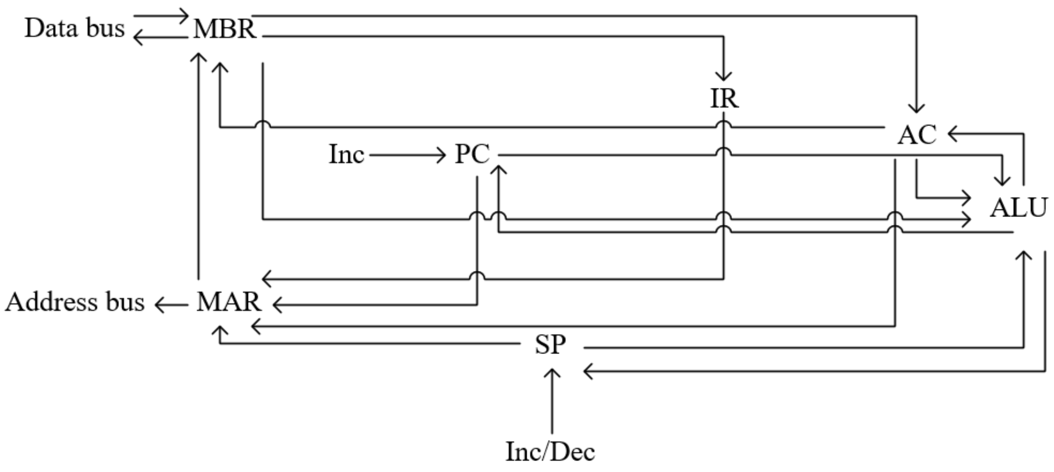
4. Use the following micro-architecture with the control signals shown below to provide the microprograms needed for the operations below. An example is shown first.

C0 = PC → MAR	C1 = MAR → address bus	C2 = memory read
C3 = memory write	C4 = Data bus → MBR	C5 = MBR → Data bus
C6 = MBR → IR	C7 = MBR → AC	C8 = AC → MBR
C9 = IR (address portion) → MAR	C10 = IR (address portion) → ALU	
C11 = MBR → MAR	C12 = AC → ALU	C13 = MBR → ALU
C14 = Inc PC	C15 = PC → ALU	C16 = ALU → PC
C17 = Inc SP	C18 = Dec SP	C19 = SP → MAR
C20 = SP → ALU	C21 = ALU → SP	C22 = ALU add
C23 = ALU subtract	C24 = ALU negation	C25 = ALU Inc
C26 = ALU → AC	C27 = AC < 0	C28 = AC == 0
C29 = AC > 0	C30 = PC → MBR	C31 = microbranch

To use C27, C28, C29, a value has to be moved to the ALU (either previously computed by for instance ALU subtract, or from the AC) and then an if-else statement follows in which, if the condition tested is true, the if clause executes, otherwise a microbranch executes.

As an example, the instruction fetch microprogram would be:

T0: C0	PC → MAR
T1: C1, C2, C4	MAR → addr bus, memory read, data bus → MBR
T2: C6, C14	MBR → IR, Inc PC



a. Write the get operand microprogram.
b. Write the microprogram for the assembly instruction ADD which adds the datum in the AC and the datum in the MBR, storing the result in the AC.
c. Write the microprogram for the assembly instruction SkipOnZero. This is the same as the MARIE instruction skipcond 01.

Answer:

a. t0: C9 (IR → MAR)
   t1: C1, C2, C4 (MAR → addr bus, memory read, data bus → MBR)

b. t0: C12, C13, C22, C26 (AC → ALU, MBR → ALU, ALU add, ALU → AC)

c. t0: C12, C27 (AC → ALU, AC < 0?)
   t1: if true C14 else C31 *microaddress* for the microprogram for instruction fetch

5. One type of microoperation is a microbranch. There are conditional microbranches and unconditional microbranches. How are unconditional microbranches used in microprograms? Provide an example machine/assembly instruction that would require a conditional microbranch.

Answer: An unconditional microbranch is used to branch from the end of one microprogram to the next one needed. This takes place going from the fetch microprogram to the decode microprogram and from the end of an instruction's microprogram (e.g., the add microprogram from 4b) to the fetch microprogram. The conditional microprogram is used within conditional instructions' microprograms (like the C27 microbranch in 4c) or based on the type of machine (assembly) instruction. For instance, some instructions requires an operand, so a conditional microbranch is used at the end of the decode microprogram to either branch to the get operand microprogram or to the specific machine instruction's microprogram. That code might look like this:

```
If IR[…] = [op code for add] then microbranch to get op
Else if IR[…] = [op code for sub] then microbranch to get op
Else if …
If IR[…] = [op code for Jump] then microbranch to Jump microprogram
Else if IR[…] = [op code for Input] then microbranch to Input microprogram
Else if…
```

6. Several upgrades were made between the 486 processor and the Pentium I to better support the pipeline. For each upgrade (see slide 22), what aspect of the pipeline did it improve, or was the improvement to some other aspect of processor performance? Skip the last upgrade (the FP error).

Answer:

Upgraded data bus – allowed larger instructions to be fetched at a time so that long instructions would be less likely to require a second fetch stage.

Superscalar – permitted execution of up to 2 instructions at a time.

Branch speculation – reduced the impact of a 3 cycle branch penalty.

Separate caches – allowed instruction fetches to proceed independent of load/store instructions.

7. For the Pentium IV architecture, answer the following questions.
   a. Assuming branches are computed in a simple integer ALU unit which takes 1 cycle to complete, how many cycles is a branch miss-speculation? How many total instructions might have been incorrectly fetched?
   b. The Pentium IV has 2 "simple" integer ALUs yet the integer ALU takes 1 cycle to execute, so why is there a need for two of them?
   c. How does pipelining of microoperations improve over pipelining of machine instructions? In spite of this, there are still stalls because of also operating at the machine instruction-level, what are these sources of possible stalls and why?

   Answers:
   a. From slide 24, the execution stage occurs after instruction fetch (3 stages), decode (2 stages) and issue (2 stages). I am assuming a branch takes 1 cycle to execute, so this would mean that 7 stages precede this stage meaning a 7 cycle branch penalty for a miss-speculation. If branches take more than 1 cycle to execute (and some branches like the loop instruction are more complicated) then this penalty increases. Given that 16 bytes are fetched at a time and the decode can decode up to 3 instructions at a time, and that up to 3 instructions can be issued per cycle, it is possible that there are more than 7 incorrectly fetched instructions that need to be

flushed from the reorder buffer and reservation stations. If we assume that the 16 byte fetch is, on average, 3 instructions, then we would have as many as 21 instructions to flush. However, what sits in the reorder buffer are microinstructions and any given machine instruction might equate to several microinstructions. So we are probably look at 10-21 microinstructions which equate to perhaps 5-10 assembly instructions.

b. Because the hardware is issuing up to 3 microinstructions per cycle, so its possible that multiple microinstructions need the same int unit and therefore having 2 int units can pay off. This can be the case if two microinstructions pertain to two different machine instructions, or if one machine instruction requires two (or more) integer operations that can take place in parallel (although I can't think of any single instruction where this would be the case).

c. All microinstructions are of the same length and each microinstruction takes exactly 1 cycle to execute (excluding FP, * and / operations). Thus, once converted to microcode, the instruction can move through the remaining portions of the pipeline at 1 cycle per stage and not cause stalls as we saw with the 486 pipeline. The CISC nature of the Intel instructions impact the pipeline in three ways. First, being variable length, the instruction fetch may require more than 1 cycle to fetch the instruction. Second, being both variable length and variable format, it may take more than 1 cycle to decode the instruction into microcode. Third, a RISC instruction would not need to be decoded into microcode and the execution of that instruction would take 1 cycle (excluding FP, * and /) but a CISC instruction might decompose into several microcode instructions which have to be issued sequentially. Data dependencies might exist between microinstructions of the same machine instruction causing further stalls.

8. Compare three ways that the Intel Pentium IV and/or iCore processors differ from a RISC-based superscalar in terms of extra hardware needed to improve performance because of the CISC-based nature of the Intel instruction set.

   Answer: There is a more aggressive branch prediction scheme used because of the length of any branch penalty. The iCore uses a multilevel branch target buffer (rather than a single level BTB or multiple branch prediction buffers). The Pentium and iCore both have a trace cache to deal with small loops and other small blocks of code so store branch behaviors. The iCore has several decoders to decode up to four instructions at a time into microcode. There is no equivalent step or hardware in a RISC architecture because microcode is not used.