Ken Shirriff's blog

Computer history, restoring vintage computers, IC reverse engineering, and whatever

Reverse-engineering the multiplication algorithm in the Intel 8086 processor

While programmers today take multiplication for granted, most microprocessors in the 1970s could only add and subtract — multiplication required a slow and tedious loop implemented in assembly code.[1] One of the nice features of the Intel 8086 processor (1978) was that it provided machine instructions for multiplication,[2] able to multiply 8-bit or 16-bit numbers with a single instruction. Internally, the 8086 still performed a loop, but the loop was implemented in microcode: faster and transparent to the programmer. Even so, multiplication was a slow operation, about 24 to 30 times slower than addition.

In this blog post, I explain the multiplication process inside the 8086, analyze the microcode that it used, and discuss the hardware circuitry that helped it out.[3] My analysis is based on reverse-engineering the 8086 from die photos. The die photo below shows the chip under a microscope. I've labeled the key functional blocks; the ones that are important to this post are darker. At the left, the ALU (Arithmetic/Logic Unit) performs the arithmetic operations at the heart of multiplication: addition and shifts. Multiplication also uses a few other hardware features: the X register, the F1 flag, and a loop counter. The microcode ROM at the lower right controls the process.

Get new posts by email:

Enter your email

Subscribe

Contact

About Ken Shirriff
Mastodon

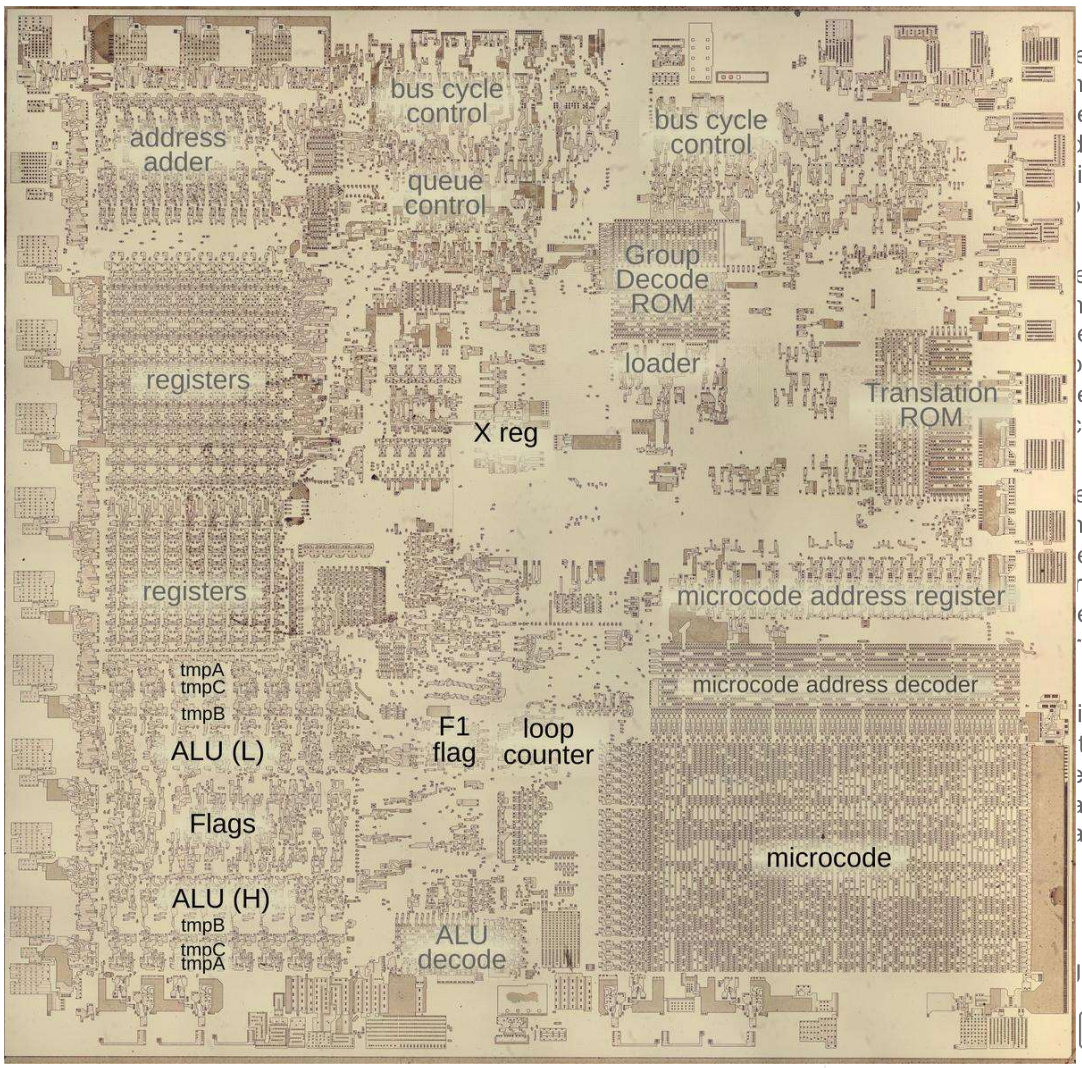
Popular Posts


Inside the Globus INK: a mechanical navigation computer for Soviet spaceflight


Reverse-engineering the multiplication algorithm in the Intel 8086 processor


How the 8086 processor determines the length of an instruction


Reverse-engineering the electronics in the Globus analog navigational computer



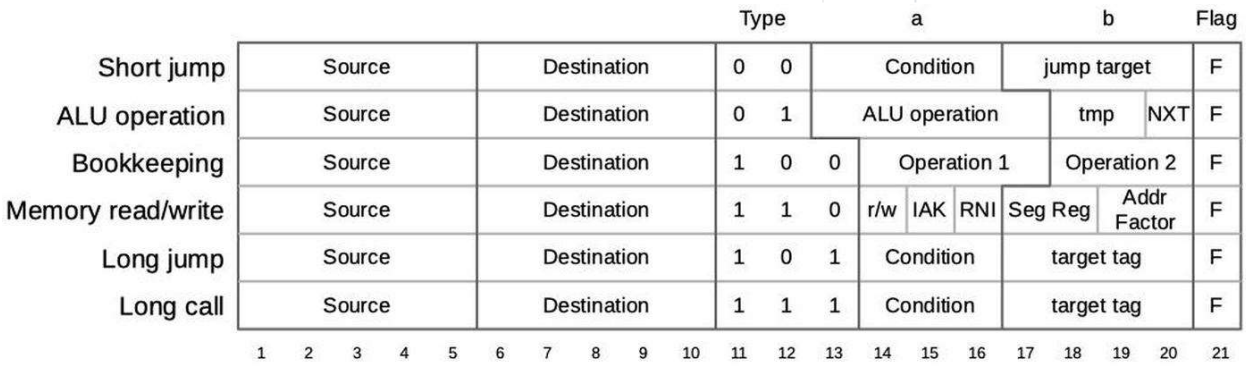
The 8086 die under a microscope, with main functional blocks labeled. This photo shows the chip with the metal and polysilicon removed, revealing the silicon underneath. Click on this image (or any other) for a larger version.

Microcode

The multiplication routines in the 8086 are implemented in microcode. Most people think of machine instructions as the basic steps that a computer performs. However, many processors (including the 8086) have another layer of software underneath: microcode. With microcode, instead of building the control circuitry from complex logic gates, the control logic is largely replaced with code. To execute a machine instruction, the computer internally executes several simpler micro-instructions, specified by the microcode. This is especially useful for a machine instruction such as multiplication, which requires many steps in a loop.

A micro-instruction in the 8086 is encoded into 21 bits as shown below. Every micro-instruction has a move from a source register to a destination register, each specified with 5 bits. The meaning of the remaining bits depends on the type field and can be anything from an ALU operation to a memory

read or write to a change of microcode control flow. Thus, an 8086 micro-instruction typically does two things in parallel: the move and the action. For more about 8086 microcode, see my microcode blog post.



The encoding of a micro-instruction into 21 bits. Based on NEC v. Intel: Will Hardware Be Drawn into the Black Hole of Copyright?

The behavior of an ALU micro-operation is important for multiplication. The ALU has three temporary registers that are invisible to the programmer: tmpA, tmpB, and tmpC. An ALU operation takes its first argument from any temporary register, while the second argument always comes from tmpB. An ALU operation requires two micro-instructions. The first micro-instruction specifies the ALU operation and source register, configuring the ALU. For instance, `ADD tmpA` to add tmpA to the default tmpB. In the next micro-instruction (or a later one), the ALU result can be accessed through the Σ register and moved to another register.

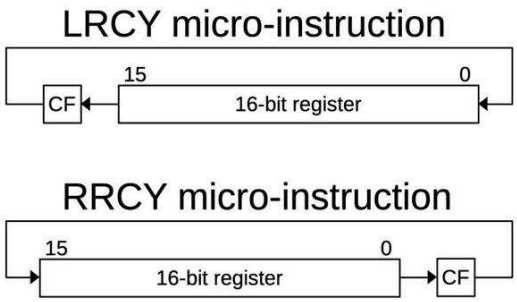
Before I get into the microcode routines, I should explain two ALU operations that play a central role in multiplication: LRCY and RRCY, Left Rotate through Carry and Right Rotate through Carry. (These correspond to the RCL and RCR machine instructions, which rotate through carry left or right.) These operations shift the bits in a 16-bit word, similar to the << and >> bit-shift operations in high-level languages, but with an additional feature. Instead of discarding the bit on the end, that bit is moved into the carry flag (CF). Meanwhile, the bit formerly in the carry flag moves into the word. You can think of this as rotating the bits while treating the carry flag as a 17th bit of the word.

Labels

6502 8008 8085 8086 8087 alto analog Apollo apple arc arduino arm beaglebone bitcoin c# cadc calculator chips css dx7 electronics f# fpga fractals genome globus haskell html5 ibm ibm1401 intel ipv6 ir java javascript math microcode oscilloscope photo power supply random reverse-engineering sheevaplug snark space spanish synth teardown theory unicode Z-80

Blog Archive

▼ 2023 (16)



The left rotate through carry and right rotate through carry micro-instructions.

These shifts perform an important part of the multiplication process since shifting can be viewed as multiplying by two. LRCY also provides a convenient way to move the most-significant bit to the carry flag, where it can be tested for a conditional jump. (This is important because the top bit is used as the sign bit.) Similarly, RRCY provides access to the least significant bit, very important for the multiplication process. Another important property is that performing RRCY on an upper word and then RRCY on a lower word will perform a 32-bit shift, since the low bit of the upper word will be moved into the high bit of the lower word via the carry bit.

Binary multiplication

The shift-and-add method of multiplication (below) is similar to grade-school long multiplication, except it uses binary instead of decimal. In each row, the multiplicand is multiplied by one digit of the multiplier. (The multiplicand is the value that gets repeatedly added, and the multiplier controls how many times it gets added.) Successive rows are shifted left one digit. At the bottom, the rows are added together to yield the product. The example below shows how 6×5 is calculated in binary using long multiplication.

```
    0110
   ×0101
    0110
   0000
  0110
 0000
00011110
```

Binary long multiplication is much simpler than decimal multiplication: at each step, you're multiplying by 0 or 1. Thus, each row is either zero or the multiplicand appropriately shifted (0110 in this case). (Unlike decimal long multiplication, you don't need to know the multiplication table.) This simplifies the hardware implementation, since each step either adds the multiplicand or doesn't. In other words, each step tests a bit of the multiplier, starting with the low bit, to determine if an add

should take place or not. This bit can be obtained by shifting the multiplier one bit to the right each step.

Although the diagram above shows the sum at the end, a real implementation performs the addition at each step of the loop, keeping a running total. Moreover, in the 8086, instead of shifting the multiplicand to the left during each step, the sum shifts to the right. (The result is the same but it makes the implementation easier.) Thus, multiplying 6×5 goes through the steps below.

```
  0101
 ×0110
 00000
 001010
 0011110
 00011110
```

Why would you shift the result to the right? There's a clever reason for this. Suppose you're multiplying two 16-bit numbers, which yields a 32-bit result. That requires four 16-bit words of storage if you use the straightforward approach. But if you look more closely, the first sum fits into 16 bits, and then you need one more bit at each step. Meanwhile, you're "using up" one bit of the multiplier at each step. So if you squeeze the sum and the multiplier together, you can fit them into two words. Shifting right accomplishes this, as the diagram below illustrates for `0xffff×0xf00f`. The sum (blue) starts in a 16-bit register called `tmpA` while the multiplier (green) is stored in the 16-bit `tmpB` register. In each step, they are both shifted right, so the sum gains one bit and the multiplier loses one bit. By the end, the sum takes up all 32 bits, split across both registers.

sum (tmpA)																multiplier (tmpC)															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	0	0	0	0	0	0	0	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0	0
0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0	0
0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0	0
0	0	0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	1	0	0
0	0	0	0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	1	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1
1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1
1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	1

The multiplication microcode

The 8086 has four multiply instructions to handle signed and unsigned multiplication of byte and word operands. These machine instructions are implemented in microcode. I'll start by describing the unsigned word multiplication, which multiplies two 16-bit values and produces a 32-bit result. The source word is provided by either a register or memory. It is multiplied by AX, the accumulator register. The 32-bit result is returned in the DX and AX registers.

The microcode below is the main routine for word multiplication, both signed and unsigned. Each micro-instruction specifies a register move on the left, and an action on the right. The moves transfer words between the visible registers and the ALU's temporary registers, while the actions are mostly subroutine calls to other micro-routines.

```
  move          action
AX → tmpC     LRCY tmpC           iMUL rmw:
M → tmpB      CALL X0 PREIMUL     called for signed multiplication
              CALL CORX           the core routine
              CALL F1 NEGATE      called for negative result
              CALL X0 IMULCOF     called for signed multiplication
tmpC → AX     JMPS X0 7
              CALL MULCOF         called for unsigned multiplication
tmpA → DX     RNI
```

The microcode starts by moving one argument AX into the ALU's temporary C register and setting up the ALU to perform a Left Rotate through Carry on this register, in order to access the sign bit. Next, it moves the second argument M into the temporary B register; M references the register or memory specified in the second byte of the instruction, the "ModR/M" byte. For a signed multiply instruction, the PREIMUL micro-subroutine is called, but I'll skip that for now. (The X0 condition tests bit 3 of the instruction, which in this case distinguishes MUL from IMUL.) Next, the CORX subroutine is called, which is the heart of the multiplication.[4] If the result needs to be negated (indicated by the F1 condition), the NEGATE micro-subroutine is called. For signed multiplication, IMULCOF is then called to set the carry and overflow flags, while MULCOF is called for unsigned multiplication. Meanwhile, the result bytes are moved from the temporary C and temporary registers to the AX and DX registers. Finally, RNI runs the next machine instruction, ending the microcode routine.

CORX

The heart of the multiplication code is the CORX routine, which performs the multiplication loop, computing the product through shifts and adds. The first two lines set up the loop, initializing the sum (tmpA) to 0. The number of loops is controlled by a special-purpose loop counter. The MAXC micro-instruction initializes the counter to 7 or 15, for a byte or word multiply respectively. The first shift of tmpC is performed, putting the low bit into the carry flag.

The loop body performs the shift-and-add step. It tests the carry flag, the low bit of the multiplicand. It skips over the ADD if there is no carry (NCY). Otherwise, tmpB is added to tmpA. (As tmpA gets shifted to the right, tmpB gets added to higher and higher positions in the result.) The tmpA and tmpC registers are rotated right. This also puts the next bit of the multiplicand into the carry flag for the next cycle. The microcode jumps to the top of the loop if the counter is not zero (NCZ). Otherwise, the subroutine returns with the result in tmpA and tmpC.

```
ZERO → tmpA   RRCY tmpC     CORX: initialize right rotate
Σ → tmpC      MAXC            get rotate result, initialize counter to max value
              JMPS NCY 8    5: top of loop
              ADD tmpA        conditionally add
Σ → tmpA                 F    sum to tmpA, update flags to get carry
              RRCY tmpA     8: 32-bit shift of tmpA/tmpC
Σ → tmpA      RRCY tmpC
Σ → tmpC      JMPS NCZ 5      loop to 5 if counter is not 0
              RTN
```

MULCOF

The last subroutine is MULCOF, which configures the carry and overflow flags. The 8086 uses the rule that if the upper half of the result is nonzero, the carry and overflow flags are set, otherwise they are cleared. The first two lines pass tmpA (the upper half of the result) through the ALU to set the zero flag for the conditional jump. As a side-effect, the other status flags will get set but these values are "undefined" in the documentation.[6] If the test is nonzero, the carry and overflow flags are set (SCOF), otherwise they are cleared (CCOF).[5] The SCOF and CCOF micro-operations were implemented solely for used by multiplication, illustrating how microcode can be designed around specific needs.

```
                PASS tmpA   MULCOF: pass tmpA through to test if zero
Σ → no dest     JMPS 12     F update flags

                JMPS Z 8    12: jump if zero
                SCOF RTN      otherwise set carry and overflow

                CCOF RTN    8: clear carry and overflow
```

8-bit multiplication

The 8086 has separate instructions for 8-bit multiplication. The process for 8-bit multiplication is similar to 16-bit multiplication, except the values are half as long and the shift-and-add loop executes 8 times instead of 16. As shown below, the 8-bit sum starts in the low half of the temporary A register and is shifted left into tmpC. Meanwhile, the 8-bit multiplier starts in the low half of tmpC and is shifted out to the right. At the end, the result is split between tmpA and tmpC.

sum (tmpA)																multiplier (tmpC)															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1
0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0

The 8086 supports many instructions with byte and word versions, using 8-bit or 16-bit arguments. In most cases, the byte and word instructions use the same microcode, with the ALU and register hardware using bytes or words based on the instruction. However, the byte- and word-multiply instructions use different registers, requiring microcode changes. In particular, the multiplier is in AL, the low half of the accumulator. At the end, the 16-bit result is returned in AX, the full 16-bit accumulator; two micro-instructions assemble the result from tmpC and tmpA into the two bytes of the accumulator, 'AL' and 'AH' respectively. Apart from those changes, the microcode is the same as the word multiply microcode discussed earlier.

```
AL → tmpC     LRCY tmpC          iMUL rmb:
M → tmpB      CALL X0 PREIMUL
              CALL CORX
              CALL F1 NEGATE
              CALL X0 IMULCOF
tmpC → AL     JMPS X0 7
              CALL MULCOF
tmpA → AH     RNI
```

Signed multiplication

The 8086 (like most computers) represents signed numbers using a format called two's complement. While a regular byte holds a number from 0 to 255, a signed byte holds a number from -128 to 127. A negative number is formed by flipping all the bits (known as the one's complement) and then adding 1, yielding the two's complement value.[7] For instance, +5 is `0x05` while -5 is `0xfb`. (Note that the top bit of a number is set for a negative number; this is the sign bit.) The nice thing about two's complement numbers is that the same addition and subtraction operations work on both signed and unsigned values. Unfortunately, this is not the case for signed multiplication, since signed and unsigned values yield different results due to sign extension.

The 8086 has separate multiplication instructions `IMUL` (Integer Multiply) to perform signed multiplication. The 8086 performs signed multiplication by converting the arguments to positive values, performing unsigned multiplication, and then negating the result if necessary. As shown above, signed and unsigned multiplication both use the same microcode, but the microcode conditionally calls some subroutines for signed multiplication. I will discuss those micro-subroutines below.

`PREIMUL`

The first subroutine for signed multiplication is `PREIMUL`, performing preliminary operations for integer multiplication. It converts the two arguments, stored in tmpC and tmpB, to positive values. It keeps track of the signs using an internal flag called `F1`, toggling this flag for a negative argument. This conveniently handles the rule that two negatives make a positive since complementing the `F1` flag twice will clear it.

This microcode, below, illustrates the complexity of microcode and how micro-operations are carefully arranged to get the right values at the right time. The first micro-instruction performs one ALU operation and sets up a second operation.

The calling code had set up the ALU to perform LRCY tmpC, so that's the result returned by Σ (and discarded). Performing a left rotate and discarding the result may seem pointless, but the important side-effect is that the top bit (i.e. the sign bit) ends up in the carry flag. The microcode does not have a conditional jump based on the sign, but has a conditional jump based on carry, so the point is to test if tmpC is negative. The first micro-instruction also sets up negation (NEG tmpC) for the *next* ALU operation.

```
Σ → no dest   NEG tmpC     PREIMUL: set up negation of tmpC
              JMPS NCY 7     jump if tmpC positive
Σ → tmpC      CF1            if negative, negate tmpC, flip F1
              JMPS 7         jump to shared code

              LRCY tmpB    7:
Σ → no dest   NEG tmpB       set up negation of tmpB
              JMPS NCY 11    jump if tmpB positive
Σ → tmpB      CF1 RTN        if negative, negate tmpB, flip F1
              RTN          11: return
```

For the remaining lines, if the carry is clear (NCY), the next two lines are skipped. Otherwise, the ALU result (Σ) is written to tmpC, making it positive, and the F1 flag is complemented with CF1. (The second short jump (JMPS) may look unnecessary, but I reordered the code for clarity.) The second half of the microcode performs a similar test on tmpB. If tmpB is negative, it is negated and F1 is toggled.

NEGATE

The microcode below is called after computing the result, if the result needs to be made negative. Negation is harder than you might expect because the result is split between the tmpA and tmpC registers. The two's complement operation (NEG) is applied to the low word, while either 2's complement or one's complement (COM1) is applied to the upper word, depending on the carry for mathematical reasons.[8] The code also toggles F1 and makes tmpB positive; I think this code is only useful for division, which also uses the NEGATE subroutine.

```
                NEG tmpC     NEGATE: negate tmpC
Σ → tmpC        COM1 tmpA F  maybe complement tmpA
                JMPS CY 6
                NEG tmpA       negate tmpA if there's no carry
Σ → tmpA        CF1          6: toggle F1 for some reason

                LRCY tmpB    7: test sign of tmpB
Σ → no dest     NEG tmpB       maybe negate tmpB
                JMPS NCY 11  skip if tmpB positive
Σ → tmpB        CF1 RTN        else negate tmpB, toggle F1
                RTN          11: return
```

IMULCOF

The IMULCOF routine is similar to MULCOF, but the calculation is a bit trickier for a signed result. This routine sets the carry and overflow flags if the upper half of the result is significant, that is, it is not just the sign extension of the lower half.[9] In other words, the top byte is not significant if it duplicates the top bit (the sign bit) of the lower byte. The trick in the microcode is to add the top bit of the lower byte to the upper byte by putting it in the carry flag and performing an add with carry (ADC) of 0. If the result is 0, the upper byte is not significant, handling the positive and negative cases. (This also holds for words instead of bytes.)

```
ZERO → tmpB   LRCY tmpC    IMULCOF: get top bit of tmpC
Σ → no dest   ADC tmpA       add to tmpA and 0 (tmpB)
Σ → no dest    F             update flags
              JMPS Z 8     12: jump if zero result
              SCOF RTN       otherwise set carry and overflow

              CCOF RTN     8: clear carry and overflow
```

The hardware for multiplication

For the most part, the 8086 uses the regular ALU addition and shifts for the multiplication algorithm. Some special hardware features provide assistance.

Loop counter

The 8086 has a special 4-bit loop counter for multiplication. This counter starts at 7 for byte multiplication and 15 for word multiplication, based on the instruction. This loop counter allows the microcode to decrement the counter, test for the end, and perform a conditional branch in one micro-operation.

The counter is implemented with four flip-flops, along with logic to compute the value after decrementing by one. The MAXC (Maximum Count) micro-instruction sets the counter to 7 or 15 for byte or word operations respectively. The NCZ (Not Counter Zero) micro-instruction has two actions. First, it performs a conditional jump if the counter is nonzero. Second, it decrements the counter.

X register

The multiplication microcode uses an internal register called the X register to distinguish between the MUL and IMUL instructions. The X register is a 3-bit register that holds the ALU opcode, indicated by bits 5–3 of the instruction.[10] Since the instruction is held in the Instruction Register, you might wonder why a separate register is required. The motivation is that some opcodes specify the type of ALU operation in the second byte of the instruction, the ModR/M byte, bits 5–3.[11] Since the ALU operation is sometimes specified in the first byte and sometimes in the second byte, the X register was added to handle both these cases.

For the most part, the X register indicates which of the eight standard ALU operations is selected (ADD, OR, ADC, SBB, AND, SUB, XOR, CMP). However, a few instructions use bit 0 of the X register to distinguish between other pairs of instructions. For instance, it distinguishes between MUL and IMUL, DIV and IDIV, CMPS and SCAS, MOVS and LODS, or AAA and AAS. While these instruction pairs may appear to have arbitrary opcodes, they have been carefully assigned. The microcode can test this bit using the X0 condition and perform conditional jumps.

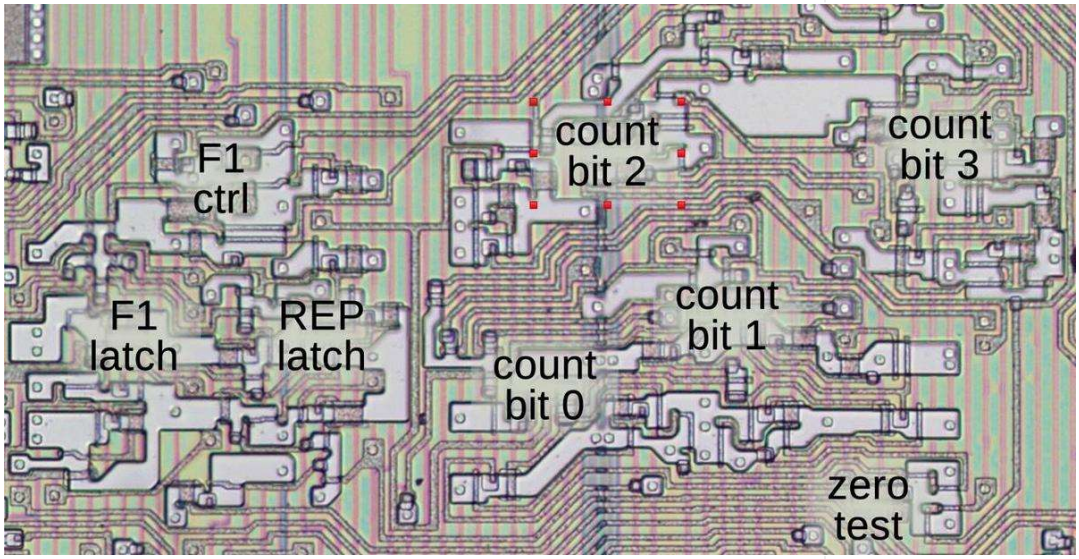
The implementation of the X register is straightforward, consisting of three flip-flops to hold the three bits of the instruction. The flip-flops are loaded from the prefetch queue bus during First Clock and during Second Clock for appropriate instructions, as the instruction bytes travel over the bus. Testing bit 0 of the X register with the X0 condition is supported by the microcode condition evaluation circuitry, so it can be used for conditional jumps in the microcode.

The F1 flag

The multiplication microcode uses an internal flag called F1,[12] which has two distinct uses. The flag keeps track of a REP prefix for use with a string operation. But the F1 flag is also used by signed multiplication and division to keep track of the

sign. The F1 flag can be toggled by microcode through the CF1 (Complement F1) micro-instruction. The F1 flag is implemented with a flip-flop, along with a multiplexer to select the value. It is cleared when a new instruction starts, set by a REP prefix, and toggled by the CF1 micro-instruction.

The diagram below shows how the F1 latch and the loop counter appear on the die. In this image, the metal layer has been removed, showing the silicon and the polysilicon wiring underneath.



The counter and F1 latch as they appear on the die. The latch for the REP state is also here.

Later advances in multiplication

The 8086 was pretty slow at multiplying compared to later Intel processors.[13] The 8086 took up to 133 clock cycles to multiply unsigned 16-bit values due to the complicated microcode loops. By 1982, the Intel 286 processor cut this time down to 21 clock cycles. The Intel 486 (1989) used an improved algorithm that could end early, so multiplying by a small number could take just 9 cycles.

Although these optimizations improved performance, they still depended on looping over the bits. With the shift to 32-bit processors, the loop time became unwieldy. The solution was to replace the loop with hardware: instead of performing 32 shift-and-add loops, an array of adders could compute the multiplication in one step. This quantity of hardware was unreasonable in the 8086 era, but as Moore's law made transistors smaller and cheaper, hardware multiplication became practical. For instance, the Cyrix Cx486SLC (1992) had a 16-bit hardware multiplier that cut word multiply down to 3 cycles. The Intel Core 2 (2006) was even faster, able to complete a 32-bit multiplication every clock cycle.

Hardware multiplication is a fairly complicated subject, with many optimizations to maximize performance while minimizing hardware.[14] Simply replacing the loop with a sequence of 32 adders is too slow because the result would be delayed while propagating through all the adders. The solution is to arrange the adders as a tree to provide parallelism. The first layer has 16 adders to add pairs of terms. The next layer adds pairs of these partial sums, and so forth. The resulting tree of adders is 5 layers deep rather than 32, reducing the time to compute the sum. Real multipliers achieve further performance improvements by splitting up the adders and creating a more complex tree: the venerable Wallace tree (1964) and Dadda multiplier (1965) are two popular approaches. Another optimization is the Booth algorithm (1951), which performs signed multiplication directly, without converting the arguments to positive values first. The Pentium 4 (2000) used a Booth encoder and a Wallace tree (ref), but research in the early 2000s found the Dadda tree is faster and it is now more popular.

Conclusions

Multiplication is much harder to compute than addition or subtraction. The 8086 processor hid this complexity from the programmer by providing four multiplication instructions for byte and word multiplication of signed or unsigned values. These instructions were implemented multiplication in microcode, performing shifts and adds in a loop. By using microcode subroutines and conditional execution, these four machine instructions share most of the microcode. As the microcode capacity of the 8086 was very small, this was a critical feature of the implementation.

If you made it through all the discussion of microcode, congratulations! Microcode is even harder to understand than assembly code. Part of the problem is that microcode is very fine-grain, with even ALU operations split into multiple steps. Another complication is that 8086 microcode performs a register move and another operation in parallel, so it's hard to keep track of what's going on. Microcode can seem a bit like a jigsaw puzzle, with pieces carefully fit together as compactly as possible. I hope the explanations here made sense, or at least gave you a feel for how microcode operates.

I've written multiple posts on the 8086 so far and plan to continue reverse-engineering the 8086 die so follow me on Twitter @kenshirriff or RSS for updates. I've also started experimenting with Mastodon recently as @kenshirriff@oldbytes.space.

Notes and references

1. Mainframes going back to ENIAC had multiply and divide instructions. However, early microprocessors took a step back and didn't supports these more complex operations. (My theory is that the decline in memory prices made it more cost-effective to implement multiply and divide in software than hardware.) The National Semiconductor IMP-16, a 16-bit bit-slice microprocessor from 1973, may be the first with multiply and divide instructions. The 8-bit Motorola 6809 processor (1978) included 8-bit multiplication but not division. I think the 8086 was the first Intel processor to support multiplication. ↩

2. The 8086 also supported division. Although the division instructions are similar to multiplication in many ways, I'm focusing on multiplication and ignoring division for this blog post. ↩

3. My microcode analysis is based on Andrew Jenner's 8086 microcode disassembly. ↩

4. I think `CORX` stands for `Core Multiply` and `CORD` stands for `Core Divide`. ↩

5. The definitions of carry and overflow are different for multiplication compared to addition and subtraction. Note that the result of a multiplication operation will always fit in the available result space, which is twice as large as the arguments. For instance, the biggest value you can get by multiplying 16-bit values is `0xffff×0xffff=0xfffe0001` which fits into 32 bits. (Signed and 8-bit multiplications fit similarly.) This is in contrast to addition and subtraction, which can exceed their available space. A carry indicates that an addition exceeded its space when treated as unsigned, while an overflow indicates that an addition exceeded its space when treated as unsigned. ↩

6. The Intel documentation states that the sign, carry, overflow, and parity flags are undefined after the `MUL` operation, even though the microcode causes them to be computed. The meaning of "undefined" is that programmers shouldn't count on the flag values because Intel might change the behavior in later chips. This thread discusses the effects of `MUL` on the flags, and how the behavior is different on the NEC V20 chip. ↩

7. It may be worth explaining why the two's complement of a number is defined by adding 1 to the one's complement. The one's complement of a number simply flips all the bits. If you take a byte value `n`, `0xff - n` is

the one's complement, since a 1 bit in n produces a 0 bit in the result.

Now, suppose we want to represent -5 as a signed byte. Adding `0x100` will keep the same byte value with a carry out of the byte. But `0x100` - 5 = (1 + 0xff) - 5 = 1 + (`0xff` - 5) = 1 + (one's complement of 5). Thus, it makes sense mathematically to represent -5 by adding 1 to the one's complement of 5, and this holds for any value. ↩

8. The negation code is a bit tricky because the result is split across two words. In most cases, the upper word is bitwise complemented. However, if the lower word is zero, then the upper word is negated (two's complement). I'll demonstrate with 16-bit values to keep the examples small. The number 257 (0x0101) is negated to form -257 (0xfeff). Note that the upper byte is the one's complement (0x01 vs 0xfe) while the lower byte is two's complement (0x01 vs 0xff). On the other hand, the number 256 (0x0100) is negated to form -256 (0xff00). In this case, the upper byte is the two's complement (0x01 vs 0xff) and the lower byte is also the two's complement (0x00 vs 0x00).

   (Mathematical explanation: the two's complement is formed by taking the one's complement and adding 1. In most cases, there won't be a carry from the low byte to the upper byte, so the upper byte will remain the one's complement. However, if the low byte is 0, the complement is 0xff and adding 1 will form a carry. Adding this carry to the upper byte yields the two's complement of that byte.)

   To support multi-word negation, the 8086's NEG instruction clears the carry flag if the operand is 0, and otherwise sets the carry flag. (This is the opposite from the above because subtractions (including NEG) treat the carry flag as a borrow flag, with the opposite meaning.) The microcode NEG operation has identical behavior to the machine instruction, since it is used to implement the machine instruction.

   Thus to perform a two-word negation, the microcode negates the low word (tmpC) and updates the flags (F). If the carry is set, the one's complement is applied to the upper word (tmpA). But if the carry is cleared, the two's complement is applied to tmpA. ↩

9. The IMULCOF routine considers the upper half of the result significant if it is not the sign extension of the lower

half. For instance, dropping the top byte of `0x0005` (+5) yields 0x05 (+5). Dropping the top byte of `0xfffb` (-5) yields 0xfb (-5). Thus, the upper byte is not significant in these cases. Conversely, dropping the top byte of `0x00fb` (+251) yields `0xfb` (-5), so the upper byte is significant. ↩

10. Curiously, the 8086 patent states that the X register is a 4-bit register holding bits 3–6 of the byte (col. 9, line 20). But looking at the die, it is a 3-bit register holding bits 3–5 of the byte. ↩

11. Some instructions are specified by bits 5–3 in the ModR/M byte rather than in the first opcode byte. The motivation is to avoid wasting bits for instructions that use a ModR/M byte but don't need a register specification. For instance, consider the instruction ADD `[BX],0x1234`. This instruction uses a ModR/M byte to specify the memory address. However, because it uses an immediate operand, it does not need the register specification normally provided by bits 5–3 of the ModR/M byte. This frees up the bits to specify the instruction. From one perspective, this is an ugly hack, while from another perspective it is a clever optimization. ↩

12. Andrew Jenner discusses the F1 flag and the interaction between REP and multiplication here. ↩

13. Here are some detailed performance numbers. The 8086 processor takes 70–77 clock cycles to multiply 8-bit values and 118–133 clock cycles to multiply 16-bit values. Signed multiplies are a bit slower because of the sign calculations: 80–98 and 128–154 clock cycles respectively. The time is variable because of the conditional jumps in the multiplication process.

    The Intel 186 (1982) optimized multiplication slightly, bringing the register word multiply down to 35–37 cycles. The Intel 286 (also 1982) reduced this to 21 clocks. The 486 (1989) used a shift-add multiply function but it had an "early out" algorithm that stopped when the remaining bits were zero, so a 16-bit multiply could take from 9 to 22 clocks. The 8087 floating point coprocessor (1980) used radix-4 multiplication, multiplying by pairs of bits at a time and either adding or subtracting. This yields half the addition cycles. The Pentium's P5 micro-architecture (1993) took the unusual approach of reusing the floating-point unit's hardware multiplier for integer multiplication, taking 10 cycles for a 32-bit multiplication. ↩

14. This presentation gives a good overview of implementations of multiplication in hardware. ↩

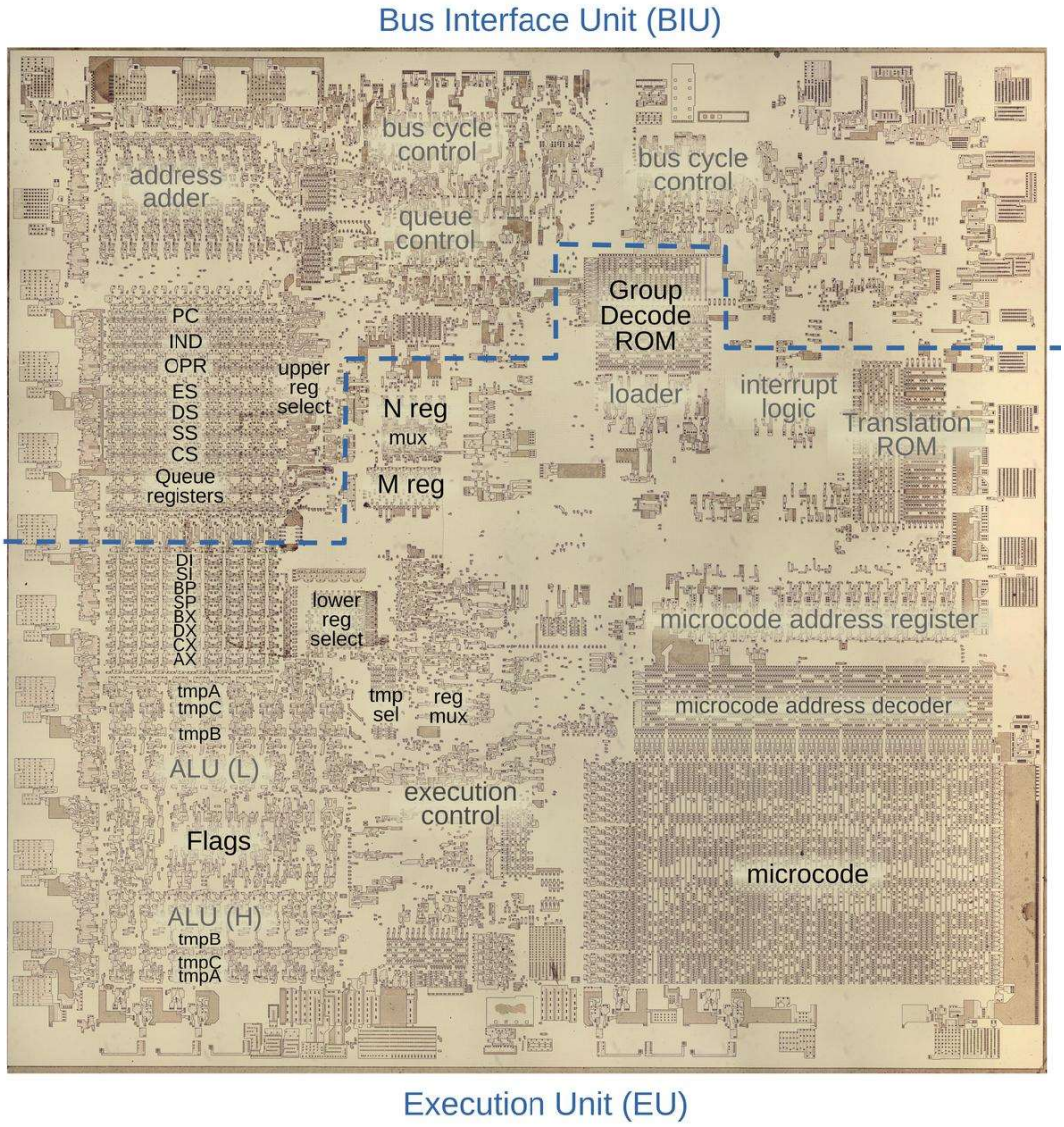
11 comments:

Labels: 8086, chips, microcode, reverse-engineering

Reverse-engineering the register codes for the 8086 processor's microcode

Like most processors, the Intel 8086 (1978) provides registers that are faster than main memory. As well as the registers that are visible to the programmer, the 8086 has a handful of internal registers that are hidden from the user. Internally, the 8086 has a complicated scheme to select which register to use, with a combination of microcode and hardware. Registers are assigned a 5-bit identifying number, either from the machine instruction or from the microcode. In this blog post, I explain how this register system works.

My analysis is based on reverse-engineering the 8086 from die photos. The die photo below shows the chip under a microscope. For this die photo, I removed the the metal and polysilicon layers, revealing the silicon underneath. I've labeled the key functional blocks; the ones that are important to this post are darker. In particular, the registers and the Arithmetic/Logic Unit (ALU) are at the left and the microcode ROM is in the lower right. Architecturally, the chip is partitioned into a Bus Interface Unit (BIU) at the top and an Execution Unit (EU) below. The BIU handles bus and memory activity as well as instruction prefetching, while the Execution Unit (EU) executes the instructions.



The 8086 die under a microscope, with main functional blocks labeled. Click on this image (or any other) for a larger version.

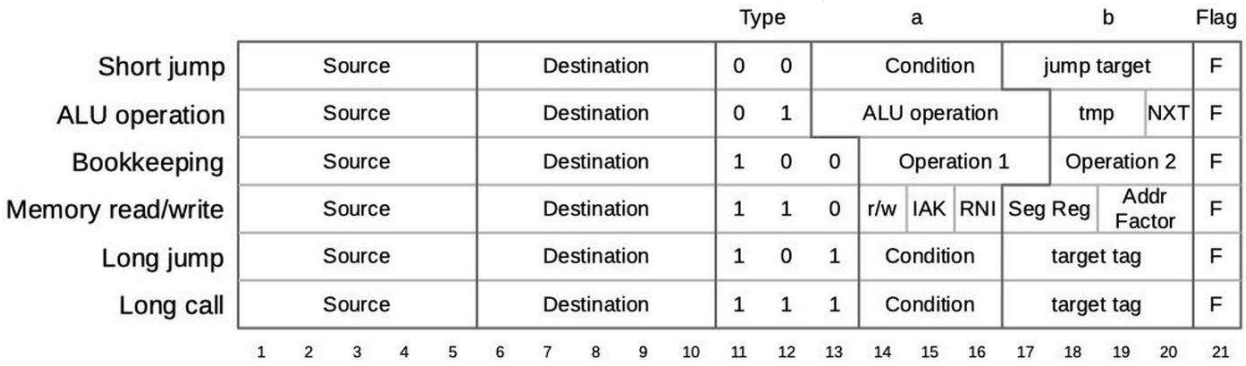
Microcode

Most people think of machine instructions as the basic steps that a computer performs. However, many processors (including the 8086) have another layer of software underneath: microcode. With microcode, instead of building the control circuitry from complex logic gates, the control logic is largely replaced with code. To execute a machine instruction, the computer internally executes several simpler micro-instructions, specified by the microcode.

The 8086 uses a hybrid approach: although it uses microcode, much of the instruction functionality is implemented with gate logic. This approach removed duplication from the microcode and kept the microcode small enough for 1978 technology. In a sense, the microcode is parameterized. For instance, the microcode can specify a generic Arithmetic/Logic Unit (ALU) operation and a generic register. The gate logic examines the

instruction to determine which specific operation to perform and the appropriate register.

A micro-instruction in the 8086 is encoded into 21 bits as shown below. Every micro-instruction has a move from a source register to a destination register, each specified with 5 bits; this encoding is the main topic of this blog post. The meaning of the remaining bits depends on the type field and can be anything from an ALU operation to a memory read or write to a change of microcode control flow. For more about 8086 microcode, see my microcode blog post.



The encoding of a micro-instruction into 21 bits. Based on NEC v. Intel: Will Hardware Be Drawn into the Black Hole of Copyright?

Let's look at how the machine instruction `XCHG AX,reg` is implemented in microcode. This instruction exchanges AX with the register specified in the low 3 bits of the instruction.[1] The microcode for this instruction consists of three micro-instructions, so the instruction takes three clock cycles. Each micro-instruction contains a move, which is the interesting part.[2] The specified register is moved to the ALU's temporary B register, the AX register is moved to the specified register, and finally the temporary B is moved to AX, completing the swap.

```
  move          action
M → tmpB                  XCHG AX,rw: move reg to tmpB
AX → M          NXT         move AX to reg, Next to last
tmpB → AX       RNI         move tmpB to AX, Run Next Instruction
```

The key part for this discussion is how `M` indicates the desired register. Suppose the instruction is `XCHG AX,DX`. The bottom three bits of the instruction are `010`, indicating DX. During the first clock cycle of instruction execution, the opcode byte is transferred from the prefetch queue over the queue bus. The `M` register is loaded with the DX number (which happens to be 26), based on the bottom three bits of the instruction. After a

second clock cycle, the microcode starts. The first micro-instruction puts M's value (26) onto the source bus and the number for tmpB (13) on the destination bus, causing the transfer from DX to tmpB. The second micro-instruction puts the AX number (24) onto the source bus and the M value (26) onto the destination bus, causing the transfer from AX to DX. The third micro-instruction puts tmpB number (13) onto the source bus and the AX number (24) onto the destination bus, causing the transfer from tmpB to AX.

Thus, the values on the source and destination bus control the data transfer during each micro-instruction. Microcode can either specify these values explicitly (as for AX and tmpB) or can specify the M register to use the register defined in the instruction. Thus, the same microcode implements all the XCHG instructions and the microcode doesn't need to know which register is involved.

The register encoding

The microcode above illustrated how different numbers specified different registers. The table below shows how the number 0-31 maps onto a register. Some numbers have a different meaning for a source register or a destination register; a slash separates these entries.

0	ES	8	AL	16	AH	24	AX
1	CS	9	CL	17	CH	25	CX
2	SS	10	DL	18	DH (M)	26	DX
3	DS	11	BL	19	BH (N)	27	BX
4	PC	12	tmpA	20	Σ/tmpAL	28	SP
5	IND	13	tmpB	21	ONES/tmpBL	29	BP
6	OPR	14	tmpC	22	CR/tmpAH	30	SI
7	Q/-	15	F	23	ZERO/tmpBH	31	DI

Most of these entries are programmer-visible registers: the segment registers are in green, the 8-bit registers in blue, and the 16-bit registers in red. Some internal registers and pseudo-registers are also accessible: IND (Indirect register), holding the memory address for a read or write; OPR (Operand register), holding the data for a read or write; Q (Queue), reading a byte from the instruction prefetch queue; ALU temporary registers A, B, and C, along with low (L) and (H) bytes; F, Flags register; Σ, the ALU output; ONES, all ones; CR, the three low bits of the microcode address; and ZERO,

the value zero. The M and N entries can only be specified from microcode, taking the place of DH and BH.

The table is kind of complicated, but there are reasons for its structure. First, machine instructions in the 8086 encode registers according to the system below. The 5-bit register number above is essentially an extension of the instruction encoding. Moreover, the AX/CX/DX/BX registers (red) are lined up with their upper-byte and lower-byte versions (blue). This simplifies the hardware since the low three bits of the register number select the register, while the upper two bits perform the byte versus word selection.[3] The internal registers fit into available spots in the table.

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

The register assignments, from MCS-86 Assembly Language Reference Guide.

The ModR/M byte

Many of the 8086 instructions use a second byte called the ModR/M byte to specify the addressing modes.[4] The ModR/M byte gives the 8086 a lot of flexibility in how an instruction accesses its operands. The byte specifies a register for one operand and either a register or memory for the other operand. The diagram below shows how the byte is split into three fields: `mod` selects the overall mode, `reg` selects a register, and `r/m` selects either a register or memory mode. For a ModR/M byte, the `reg` and the `r/m` fields are read into the N and M registers respectively, so the registers specified in the ModR/M byte can be accessed by the microcode.

mod		reg			r/m		
7	6	5	4	3	2	1	0

Let's look at the instruction `SUB AX,BX` which subtracts BX from AX. In the 8086, some important processing steps take place before the microcode starts. In particular, the "Group Decode ROM" categorizes the instruction into over a dozen categories that affect how it is processed, such as instructions that are implemented without microcode, one-byte instructions,

or instructions with a ModR/M byte. The Group Decode ROM also indicates the structure of instructions, such as instructions that have a W bit selecting byte versus word operations, or a D bit reversing the direction of the operands. In this case, the Group Decode ROM classifies the instruction as containing a D bit, a W bit, an ALU operation, and a ModR/M byte.

Based on the Group Decode ROM's signals, fields from the opcode and ModR/M bytes are extracted and stored in various internal registers. The ALU operation type (SUB) is stored in the `ALU opr` register. The ModR/M byte specifies BX in the `reg` field and AX in the `r/m` field so the `reg` register number (BX, 27) is stored in the N register, and the `r/m` register number (AX, 24) is stored in the M register.

Once the preliminary decoding is done, the microcode below for this ALU instruction is executed.[5] There are three micro-instructions, so the instruction takes three clock cycles. First, the register specified by M (i.e. AX) is moved to the ALU's temporary A register (`tmpA`). Meanwhile, `XI` configures the ALU to perform the operation specified by the instruction bits, i.e. SUB. The second micro-instruction moves the register specified by N (i.e. BX) to the ALU's `tmpB` register. The last micro-instruction stores the ALU's result (Σ, number 20) in the register indicated by M (i.e. AX).

move	action	
M → tmpA	XI tmpA	***ALU rm↔r**: AX to tmpA*
N → tmpB	NXT	*BX to tmpB*
Σ → M	RNI F	*result to AX, update flags*

One of the interesting features of the 8086 is that many instructions contain a D bit that reverses the direction of the operation, swapping the source and the destination. If we keep the ModR/M byte but use the SUB instruction with the D bit set, the instruction becomes `SUB BX,AX`, subtracting AX from BX, the opposite of before. (Swapping the source and destination is more useful when one argument is in memory. But I'll use an example with two registers to keep it simple.) This instruction runs exactly the same microcode as before. The difference is that when the microcode accesses M, due to the direction bit it gets the value in N, i.e. BX instead of AX. The access to N is similarly swapped. The result is that AX is subtracted from BX, and the change of direction is transparent to the microcode.

The M and N registers

Now let's take a closer look at how the M and N registers are implemented. Each register holds a 5-bit register number, expanded from the three bits of the instruction. The M register is loaded from the three least significant bits of the instruction or ModR/M byte, while the N register is loaded with bits three through five. Most commonly, the registers are specified by the ModR/M byte, but some instructions specify the register in the opcode. [6]

The table below shows how the bits in the instruction's opcode or ModR/M byte (i5, i4, i3) are converted to a 5-bit number for the N register. There are three cases: a 16-bit register, an 8-bit register, and a segment register. The mappings below may seem random, but they result in the entries shown in the 5-bit register encoding table earlier. I've colored the entries so you can see the correspondence.

Mode	4	3	2	1	0
16-bit reg	1	1	i5	i4	i3
8-bit reg	i5	i5'	0	i4	i3
segment reg	0	0	0	i4	i3

I'll go through the three cases in more detail. Many 8086 instructions have two versions, one that acts on bytes and one that acts on words, distinguished by the W bit (bit 0) in the instruction. If the Group Decode ROM indicates that the instruction has a W bit and the W bit is 0, then the instruction is a byte instruction. [7] If the instruction has a ModR/M byte and the instruction operates on a byte, the N register is loaded with the 5-bit number for the specified byte register. This happens during "Second Clock", the clock cycle when the ModR/M byte is fetched from the instruction queue. The second case is similar; if the instruction operates on a word, the N register is loaded with the number for the word register specified in the ModR/M byte.

The third case handles a segment register. The N register is loaded with a segment register number during Second Clock if the Group Decode ROM indicates the instruction has a ModR/M byte with a segment-register field (specifically the segment register MOV instructions). A bit surprisingly, a segment register number is also loaded during First Clock. This supports the PUSH and POP segment register instructions, which have the segment register encoded in bits 3 and 4 of the opcode. [8]

The table below shows how the bits are assigned in the M register, which uses instruction bits `i2`, `i1`, and `i0`. The cases are a bit more complicated than the N register. First, a 16-bit register number is loaded from the opcode byte during First Clock to support instructions that specify the register in the low bits. During Second Clock, this value may be replaced.

For a ModR/M byte using register mode, the M register is reloaded with the specified 8-bit or a 16-bit register, depending on the byte mode signal described earlier. However, for a ModR/M byte that uses a memory mode, the M register is loaded with OPR (Operand), the internal register that holds the word that is read or written to memory.

Mode	4	3	2	1	0
16-bit reg	1	1	i2	i1	i0
8-bit reg	i2	i2'	0	i1	i0
OPR	0	0	1	1	0
AX/AL	byte'	1	0	0	0
convert to 8-bit	m2	m2'	0	m1	m0

Many instructions use the AX or AL register, such as the ALU immediate instructions, the input and output instructions, and string instructions. For these, the Group Decode ROM triggers the AX or AL register number specifically to be loaded into the M register during Second Clock. The top bit is set for a word operation and cleared for a byte operation providing AX or AL as appropriate.

The final M register case is a bit tricky. For an immediate move instruction such as `MOV BX,imm`, bit 3 switches between a byte and a word move (rather than bit 0), because bits 2-0 specify the register. Unfortunately, the Group Decode ROM outputs aren't available during First Clock to indicate this case. Instead, M is loaded during First Clock with the assumption of a 16-bit register. If that turns out to be wrong, the M register is converted to an 8-bit register number during Second Clock by shuffling a few bits.

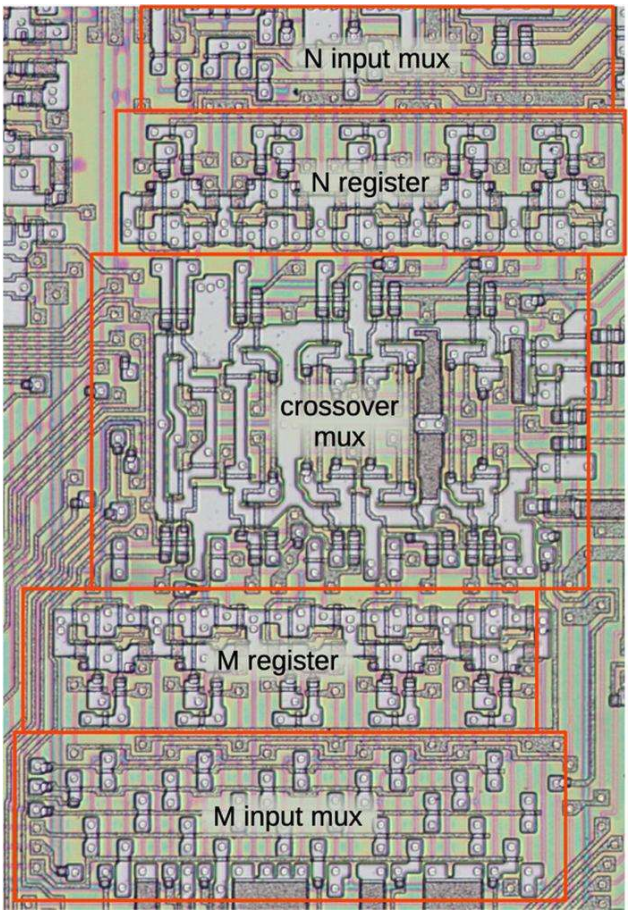
Producing the source and destination values

There are three cases for the number that goes on the source or destination buses: the register number can come from the micro-instruction, the value can come from the M or N register as specified in the micro-instruction, or the value can come from the M and N register with the roles swapped by the D bit.

(Note that the source and destination can be different cases and are selected with separate circuitry.)

The first case is the default case, where the 5 bits from the micro-instruction source or destination specify the register directly. For instance, in the micro-instruction `tmpB→AX`, the microcode knows which registers are being used and specifies them directly.

The second and third cases involve more logic. Consider the source in `M→tmpB`. For an instruction without a `D` bit, the register number is taken from `M`. Likewise if the `D` bit is `0`. But if the instruction uses a `D` bit and the `D` bit is `1`, then the register number is taken from `N`. Multiplexers between the `M` and `N` registers select the appropriate register to put on the bus.



The M and N registers as they appear on the die. The metal layer has been removed from this image to show the silicon and polysilicon underneath.

The diagram above shows how the `M` and `N` register circuitry is implemented on the die, with the `N` register at the top and the `M` register below. Each register has an input multiplexer that implements the tables above, selecting the appropriate 5 bits depending on the mode. The registers themselves are implemented as dynamic latches driven by the clock. In the middle, a crossover multiplexer drives the source and

destination buses, selecting the M and N registers as appropriate and amplifying the signals with relatively large transistors. The third output from the multiplexer, the bits from the micro-instruction, is implemented in circuitry physically separated and closer to the microcode ROM.

The register selection hardware

How does the 5-bit number select a register? The 8086 has a bunch of logic that turns a register number into a control line that enables reading or writing of the register. For the most part, this logic is implemented with NOR gates that match a particular register number and generate a select signal. The signal goes through a special bootstrap driver to boost its voltage since it needs to control 16 register bits.

The 8086 registers are separated into two main groups. The "upper registers" are in the upper left of the chip, in the Bus Interface Unit. These are the registers that are directly involved with memory accesses. The "lower registers" are in the lower left of the chip, in the Execution Unit. From bottom to top, they are AX, CX, DX, BX, SP, BP, SI, and DI; their physical order matches their order in the instruction set.[9] A separate PLA (Programmable Logic Array) selects the ALU temporary registers or flags as destination. Just below it, a PLA selects the source from ALU temporary registers, flags, or the ALU result (Σ).[10] I've written about the 8086's registers and their low-level implementation here if you want more information.

Some history

The 8086's system of selecting registers with 3-bit codes originates with the Datapoint 2200,[11] a desktop computer announced in 1970. The processor of the Datapoint 2200 was implemented with a board of TTL integrated circuits, since this was before microprocessors. Many of the Datapoint's instructions used a 3-bit code to select a register, with a destination register specification in bits 5-3 of the instruction and a source register in bits 2-0. (This layout is essentially the same as in 8086 instructions and the ModR/M byte.)[12] The eight values of this code selected one of 7 registers, with the eighth value indicating a memory access. Intel copied the Datapoint 2200 architecture for the 8008[13] microprocessor (1972) and cleaned it up for the 8080 (1974), but kept the basic instruction layout and register/memory selection bits.

The 8086's use of a numbering system for all the registers goes considerably beyond this pattern, partly because its registers function both as general-purpose registers and special-purpose registers.[14] Many instructions can act on the AX, BX, etc. registers interchangeably, treating them as

general-purpose registers. But these registers each have their own special purposes for other instructions, so the microcode must be able to access them specifically. This motivates the 8086's approach where registers can be treated as general-purpose registers that are selected from instruction bits, or as special-purpose registers selected by the microcode.

The Motorola 68000 (1979) makes an interesting comparison to the 8086 since they were competitors. The 68000 uses much wider microcode (85-bit microinstructions compared to 21 bits in the 8086). It has two main internal buses, but instead of providing generic source and destination transfers like the 8086, the 68000 has a much more complicated system: about two dozen microcode fields that connect registers and other components to the bus in various ways.[15]

Conclusions

Internally, the 8086 represents registers with a 5-bit number. This is unusual compared to previous microprocessors, which usually selected registers directly from the instruction or control circuitry. Three factors motivated this design in the 8086. First, it used microcode, so a uniform method of specifying registers (both programmer-visible and internal) was useful. Second, being able to swap the source and destination in an instruction motivated a level of indirection in register specification, provided by the M and N registers. Finally, the flexibility of the ModR/M byte, in particular supporting byte, word, and segment registers, meant that the register specification needed 5 bits.

I've written multiple posts on the 8086 so far and plan to continue reverse-engineering the 8086 die so follow me on Twitter @kenshirriff or RSS for updates. I've also started experimenting with Mastodon recently as @oldbytes.space@kenshirriff.

Notes and references

1. As an aside, the NOP instruction (no operation) in the 8086 is really XCHG AX,AX. Exchanging the AX register with itself accomplishes nothing but takes 3 clock cycles. ↩

2. The action part of the micro-instructions indicates the second-last micro-instruction (NXT, next) and the last (RNI, Run Next Instruction), so execution of the next machine instruction can start. ↩

3. Note that register #18 can refer both to DH and the destination register. This doesn't cause a conflict because it refers to DH if loaded from the instruction, and

refers to the destination register if specified in the micro-instruction. The only issue is that a micro-instruction can't refer to the DH register explicitly (or the BH register similarly). This restriction isn't a problem because the microcode never needs to do this. ↩

4. I discuss the 8086's ModR/M byte in detail here. ↩

5. The microcode listings are based on Andrew Jenner's disassembly. I have made some modifications to (hopefully) make it easier to understand. ↩

6. There are a few instructions that specify a register in the opcode rather than the ModR/M byte. For 16-bit registers, the INC, DEC, XCHG, PUSH, and POP instructions specify the register in the low three bits of the opcode. The MOV immediate instructions specify either an 8-bit or 16-bit register in the low three bits. On the other hand, the segment is specified by bits 3 and 4 of the segment prefixes, PUSH, and POP instructions. ↩

7. A few instructions only have byte versions (DAA, AAA, DAS, AAS, AAM, AAD, XLAT). This is indicated by a Group Decode ROM output and forces instruction execution into byte mode. Thus, these instructions would load a byte register into N, but since these instructions don't have a register specification, it doesn't matter. ↩

8. The segment prefixes use the same instruction bits (3 and 4) as PUSH and POP to select the segment register, so you might expect the prefixes to also load the N register. However, the prefixes are implemented in hardware, rather than microcode. Thus, they do not use the N register and the N register is not loaded with the segment register number. ↩

9. You might wonder why the BX register is out of sequence with the other registers, both physically on the chip and in the instruction set. The 8086 was designed so 8080 assembly code could be translated to 8086 code. Originally, the 8086 registers had different names: XA, BC, DE, HL, SP, MP, IJ, and IK. The first four names matched the registers in the Intel 8080 processor, while MP was Memory Pointer and IJ and IK were Index registers. However, when the 8086 was released the registers were given names that corresponded to their functions in the 8086, abandoning the 8080 names. XA became the Accumulator AX, The BC register was used for counting, so it became the Count register CX. The DE

register was a data register, so it became the Data register DX. The HL register was used as a base for memory accesses, so it became the Base register BX. The result is that the BX register ended up last.

A program CONV-86 allowed 8080 assembly programs to be translated into 8086 assembly programs, with 8080 registers replaced with the corresponding 8086 registers. The old 8086 register names can be seen in the 8086 patent, while the Accumulator, Base, Count, Data names are in the MCS-86 Assembly Language Reference Guide. See also this Stack Exchange discussion. ↩

10. The all-ones source doesn't have any decoding; the ALU bus is precharged to the high state, so it is all ones by default. ↩

11. The system of using bit fields in instructions to select registers is much older, of course. The groundbreaking IBM System/360 architecture (1964), for instance, used 4-bit fields in instructions to select one of the 16 general-purpose registers. ↩

12. Note that with this instruction layout, the instruction set maps cleanly onto octal. The Datapoint 2200 used octal to describe the instruction set, but Intel switched to hexadecimal for its processors. Hexadecimal was becoming more popular than octal at the time, but the move to hexadecimal hides most of the inherent structure of the instructions. See x86 is an octal machine for details. ↩

13. The Datapoint manufacturer talked to Intel and Texas Instruments about replacing the board of chips with a single processor chip. Texas Instruments produced the TMX 1795 microprocessor chip and Intel produced the 8008 shortly after, both copying the Datapoint 2200's architecture and instruction set. Datapoint didn't like the performance of these chips and decided to stick with a TTL-based processor. Texas Instruments couldn't find a customer for the TMX 1795 and abandoned it. Intel, on the other hand, sold the 8008 as an 8-bit microprocessor, creating the microprocessor market in the process. Register selection in these processors was pretty simple: the 3 instruction bits were decoded into 8 select lines that selected the appropriate register (or memory). Since these processors had hard-coded control instead of microcode, the control circuity generated other register selection lines directly. ↩

14. While the 8086 has eight registers that can be viewed as general-purpose, they all have some specific purposes. The AX register acts as the accumulator and has several special functions, such as its use in the XCHG (Exchange) operation, I/O operations, multiplication, and division. The BX register has a special role as a base register for memory accesses. The CX register acts as the counter for string operations and for the JCXZ (Jump if CX Zero) instruction. The DX register can specify the port for I/O operations and is used for CWD (Convert Word to Doubleword) and in multiplication and division. The SP register has a unique role as the stack pointer. The SI and DI registers are used as index registers for string operations and memory accesses. Finally, the BP register has a unique role as base pointer into the stack segment. On the 8-bit side, AX, BX, CX, and DX can be accessed as 8-bit registers, while the other registers cannot. The 8-bit AL register is used specifically for XLAT (Translate) while AH is used for the flag operations LAHF and SAHF. Thus, the 8086's registers are not completely orthogonal, and each one has some special cases, often for historical reasons. ↩

15. Another way of looking at the Motorola 68000's microcode is that the register controls come from "horizontal" microcode, a micro-instruction with many bits, and fields that control functional elements directly. The 8086's microcode is more "vertical"; the micro-instructions have relatively few bits and the fields are highly encoded. In particular, the 8086's source and destination register fields are highly encoded, while the 68000 has fields that control the connection of an individual register to the bus. ↩

2 comments:

Labels: 8086, chips, microcode, reverse-engineering

Home Older Posts