Lecture 2 Computer Organization and Design   DKP/Fall 2020

General Introduction (continued)

Let's look inside the processor. I'll redraw the block diagram of the whole uniprocessor, this time slightly differently.

```
  Memory              Processor           Peripheral devices
+---------------+   +---------------+   +---------------+
|               |   |               |   |               |
|               |   | Registers     |   | Input devices |
| Memory        |   |               |   |               |
| hierarchy     |---| PC        ALU |---|- - - - - - - -|
|               |   |               |   |               |
|               |   | Control unit/ |   | Output devices|
|               |   | Datapath      |   |               |
|               |   |               |   |               |
+---------------+   +---------------+   +---------------+
```

This picture is a little less physical (more abstract), but we can still imagine the point-to-point interconnect on the left, and the shared I/O bus on the right. Looking carefully, we can make out four blocks.

1) The processor (formerly known as the CPU) contains

i) an arithmetic-logic unit (ALU) that performs arithmetic and logical operations,

ii) a file of registers whose main function is to serve as high-speed storage of operands,

iii) a control unit/datapath (the two are inseparable) that interprets instructions, and causes them to be executed, and

iv) a program counter (PC) that lives inside the datapath and indicates the memory address of the next instruction to be fetched. We could say that the _datapath_ (now more commonly called "pipeline" because all modern datapaths are pipelined) does the work of executing instructions, while the _control unit_ specifies how this work should be carried out. You may think of the control unit as an abstract "control program" that has been burned into the silicon chip; we call such a program _hard-wired_. This is the current solution.

Previously, the control unit was an actual (software) program that ran on a smaller implementing processor _inside_ the "real" processor, i.e., the one visible to the compiler, the assembler, and the programmer. The old term for this was "microprogramming".

Note: Modern terminology would distinguish

i) the pipeline,
ii) the register file,
iii) the individual functional units, and
iv) the on-chip caches.

2) A memory that stores instructions, data, and intermediate and final results. Memory is now implemented as a hierarchy. Memory is typically byte addressed.

3) A set of peripheral _input_ devices that _send_ (transmit) data and instructions---sometimes from themselves, sometimes from the outside world--- to the memory. The disk (a storage device) functions as an input device, and so do I/O peripherals such as the keyboard and the network-interface chip.

4) A set of peripheral _output_ devices that _receive_ (transmit) final results and messages from the memory---sometimes to themselves, sometimes to the outside world. Again, the disk functions as an output device, but so do I/O peripherals such as the monitor ("screen") and the printer.

Recall the fetch-execute cycle in more detail.

1) The fetch engine ("f-box") in the pipeline fetches an instruction from memory as specified by the address stored in PC.

2) The PC is unconditionally incremented by the length of an instruction in bytes---assuming, as we do, that memory is byte addressed. Unless otherwise specified, we will consider all instructions to be encoded in 32 bits. If the fetched instruction is a conditional or unconditional branch, further modification of the PC may take place, by some other box, at a later time.

3) The decode engine ("d-box") in the pipeline decodes the instruction, in conjunction with the control unit. This is the first time we know what kind of instruction we have fetched. (We will augment the d-box later).

4) For a register-register instruction, the execution engine ("x-box") in the pipeline executes the instruction's _operation_. But this is only one of several types of instruction. For example, we could equally well have a load instruction transferring data from a memory location to a register, a store instruction transferring data from a register to a memory location, or a conditional-branch instruction. So, we need to add at least a load-store engine ("m-box") to the pipeline. Actually, we need to add one more. We will list all pipeline engines later. (They are called _pipeline boxes_).

5) And repeat until shutdown (or whatever).

Note: The details about the exact sequencing of actions will make sense much later when we consider instruction _pipelining_ in the control unit/datapath.

In 2020, this picture is somewhat dated. For example, we don't have ALUs anymore; they have been replaced by sets of functional units. Portions of the memory hierarchy (specifically the L1$, L2$, and L3$) have been integrated into the processor chip. (These are the processor _caches_). And multicore means that several processors (now called _cores_), each with its private low-level cache(s), have been integrated onto a single "processor" chip.

Finally, although this is more advanced, sometimes we deviate from strict program order deep inside the pipeline. This deviation is hidden from the running program. This advanced technique is called _dynamic instruction scheduling_, and conceptually makes use of dataflow ideas, thus partially breaking away from the von Neumann model of pure control-flow scheduling. This is the main idea in RISC 2.0.

We haven't said much about processor performance.

Historically, the two factors with the greatest impact on performance have been

i) increases in clock frequency, and
ii) increases in the number of transistors (hence, gates) that can be packed onto a single chip. Peak performance is obviously affected by the _product_ "number of logic transistors-Hz". How have these two factors evolved over the years?

Moore's Law

____

>From 1971 to 2002, clock speeds increased at an exponential rate (roughly doubling every 2.5 years). After 2003, the frequencies stabilize in the 3-GHz range (otherwise the chips would burn up---unless you cooled them with Freon).

But the other factor is (or rather was) still going strong. The number of transistors that can be put on a chip rose at the same pace (or better) as the clock frequency over this period, but without any leveling off in 2003.

Moore's law has had several incarnations:

- Initially, it was about the exponential increase in the number of _memory transistors_ per square centimeter, and per dollar (Moore's memory law).
- Later, it was about a similar exponential increase in the number of _logic transistors_ per square centimeter, and per dollar (Moore's processor law).
- And, for a while, it was about the _combined_ exponential increase in both the number of logic transistors and the clock frequency. (Strictly speaking, it was about the coexistence of Moore's law and Dennard scaling, according to which smaller transistors use less power). Number of logic transistors-Hz isn't everything, but it's obviously important for performance.
- Since 2003, Moore's law has reverted back to mean an exponential increase in number of logic transistors per square centimeter, and hence in the number of logic transistors per processor chip. How long can Moore's law continue? That's one of the $64,000 questions.
- Since 2014, it has been going through a rough patch. Some even claim that Moore's law has been repealed. But we do have quite a number of transistors per chip now. What is the best way to use them?

It is easy to understand that there might be one Moore's law for memory, and another Moore's law for processors. But why does clock frequency enter the picture and then suddenly depart? Logic transistors were, and still are, getting smaller. For a time, there was a concurrent phenomenon: _Dennard scaling_. When Dennard scaling held, the following was true: as transistors got smaller, the power density was constant---so if there was a reduction in a transistor's linear size by two, the power it used fell by four (with voltage and current both falling by two). As a result, the total chip power for a given chip-area size stayed the same from one VLSI-process generation to the next. At the same time, feature sizes shrank, transistor count doubled, and the clock frequency increased by 40% every two years. However, when feature sizes dropped below 65 nm, Dennard scaling could no longer be sustained, because of the exponential growth of the leakage current. In short, although today we still have (or had) an exponentially increasing number of logic transistors per processor chip, we can no longer afford the power to turn them all on, and can no longer tolerate the heat of clocking them faster.

Again, we have billions and billions of transistors, but can no longer afford the power to turn them all on. Multicore is a different path to steadily increasing performance that deliberately underclocks cores to stay within the power budget. Multicore has the potential to restore _power efficiency_, which is the number of operations/second per watt. We're still trying to figure out to what extent multicore is performance scalable. Current implementations are disappointing. People just don't see that you need a large number of exceptionally low-power, low-performance cores, and a large number of lightweight threads to keep them busy. Multicore is a win only if you take thread-level parallelism seriously. The architecture must enable programs to be decomposed into massive numbers of fine-grained threads and must also use the _simplest possible_ cores for area and power efficiency.

The broader significance for the future of computing is this. It used to be that sequential processors had steadily increasing performance; that let programmers carry on with business as usual. Now it is the case that only parallel processors will have steadily increasing performance; programmers will have to get off their asses and learn to program the new machines.

I personally find it unlikely that programming systems, such as Google's MapReduce, will allow parallel-oblivious programmers to survive in the new era. Using a tool to program for you is not exactly a comprehensive solution to the problem of enabling general-purpose parallel programming.

Speaking as a computer architect, it appears that memory is the critical bottleneck for multicore: until we have major increases in memory bandwidth, multicore growth will be stunted. Indeed, even the highest-level shared cache is a bottleneck, because of the unrelenting contention for it among the cores on the chip. Note: GPUs need high memory bandwidth; CPUs need low memory latency. As a general rule, when too many threads share the same cache, the performance crashes. Or, possibly, RISC 2.0 is the critical bottleneck.

Again, Intel and other manufacturers capped their clock frequencies at their 2003 level. IBM has been a bit more daring.

Exercise:

Assume processor performance is proportional to the product of the number of logic transistors per square centimeter times the clock frequency in Hz. Consider a span of 16 years.

Scenario 1: The number of logic transistors doubles every 1.9 years; the clock frequency doubles every 2.5 years.

Scenario 2: The number of logic transistors doubles every 2.1 years; the clock frequency stays constant after 8 years of doubling as in Scen. 1.

What is the relative performance improvement?

Answer:

After 16 years, the processors in scenario 1 are a bit more than 16 times faster than the processors in scenario 2 (more precisely, 2^4.002 times faster).

Amdahl's Law

____

Finally, I must tell you about a famous law about how efforts to improve performance (or reduce power or whatever) sometimes lead to disappointments, or at least to surprises. This is _Amdahl's law_, introduced by Gene Amdahl in 1967. Consider a program with one portion that is perfectly sequential, and another perfectly parallel portion that can be made as parallel as we like. Suppose the sequential portion accounts for 5% of the run time when the program is run sequentially. What happens if the program is run on a 1000-core processor?

We draw little diagrams. If you have any sense, you will avoid formulas.

```
  5  +   95  =  100
 /1   /1000
 __  _____    ___
  5  + 0.1  =     5.1   speed up = 19.6 (= 100/5.1)
```

The speedup isn't 1,000; it's barely 20.

In general terms, Amdahl's law says that optimizing one part of the system that contributes the fraction 'p' of the quantity being minimized can yield _at best_ an improvement of 1/(1 - p). Example: When p = 0.95, the best result is a factor of 20. This works for time, power, etc.

Example: Gene Amdahl originally observed that the less parallel portion of a program can limit performance on a parallel computer. Thus, one might consider reserving part of a multicore chip for a larger core specialized in single-thread performance.

Suppose we ignore this suggestion and build 100 identical cores. Program P has portion A that uses 10% of the sequential time, and gets no parallel speedup, and portion B that uses 90% of the sequential time and gets a parallel speedup equal to the number of cores. What is the speedup run time of P on this parallel computer?

```
 10  +  90  =  100
 /1   /100
 __  ____    ___
 10 +  0.9  = 10.9    speed u = 9.2
```

Now, let us cannibalize the resources required to build 10 cores to build one larger core that runs single threads twice as fast, leaving 90 cores of the original design. Program P is the same, with its 10%/90% split. What is the speedup run time of P on this new parallel computer?

```
 10  +  90  =  100
 /2   /90
 __  ___    ___

  5  +  1  =   6    su = 16.7
```

Example: In a 100-watt sequential circuit, the combinational logic dissipates 20 watts, while the (clocked) state elements dissipate 80 watts. In the combinational logic, power is proportional to the voltage, but in the state elements, power is proportional to the _square_ of the clock frequency.

(This invariant is just made up; it is fake news).

The designers propose to reduce the voltage by a factor of 8 (to help the combinational logic), and the clock frequency by a factor of 10 (to help the state elements). After the change, how much power is dissipated by the circuit?

```
 20  +  80  =  100 watts

 /8   /100

 __   ____     ___


2.5  + 0.8  =  3.3 watts
```

New plan. Starting from the original circuit, the designers now propose to reduce the voltage by a factor of 5, and the clock frequency by a factor of 15. After the change, how much power is dissipated by the circuit?

```
 20  +   80  =  100 watts

 /5    /225

 __    ____     ___

  4  + 0.36  = 4.36 watts
```

Exercise: On a uniprocessor, perfectly serial portion A of program P consumes 25 s, while perfectly parallel portion B consumes 75 s, for a total uniprocessor run time of 100 s. On a 1,000-P multiprocessor, however, program P's run time falls to 25 + 0.075 = 25.075 s. How many processors are required to achieve at least 75% of the 1,000-P speedup?

Ans: 75% of the speedup translates into a contribution of

4/3 * 25.075 = 33.433 - 25 = 8.433 s

from portion B. 8.9 processors are enough for this (8.8932806), but that's crazy.

Nine processors gives us 8 1/3 s as B's contribution, which is less than 8.433 s, so nine processors is the right answer.

At present, I don't plan to cover in any detail two low-level implementation topics. At the lowest level, there is _circuit design_. Here, the key words are "wires", "transistors", "resistors", "diodes", and so on; this is the domain of electrical engineers. The next level up is _logic design_. This is the level at which logic gates and wires are put together to build combinational circuits such as ALUs and PLAs, and at which stable-storage primitives such as flip-flops and latches are combined to implement registers and hard-wired control. We will do _some_ logic design.

Amdahl's Law

Examples:

Floating Point instructions improved to run twice as fast, but only 10% of all executed instructions are Floating Point:

$$S = \frac{1}{\left(1-f+\frac{f}{F}\right)} = \frac{1}{0.9+\frac{0.1}{2}} = \frac{1}{0.95} = 1.053$$

Signed and Unsigned Numbers

____

Computers need to represent numbers. Since computers are electrical machines, it is natural to represent numbers in hardware as a series of high and low electronic signals. This gives us only two digits, 0 and 1, so these numbers are called _binary numbers_. A single binary digit is of course a _bit_.

We will work mostly in base 2 and base 16, but numbers may be represented in any base. If 'd' is the value of the i_th digit, the contribution to the number from that digit position is d * base^i, where 'i' starts at 0 and increases from right to left.

Mathematicians have a set of numbers they call the _natural numbers_ (i.e., the nonnegative integers including 0). The use of binary bit patterns to represent natural numbers follows what is called _natural-number semantics_. (Computer jargon for natural number is "unsigned number").

Example: In natural-number semantics, the bit pattern 1011 represents

(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) = 11.

The bit corresponding to exponent 0 is the _least significant bit_. The bit corresponding to the largest exponent is the _most significant bit_.

Suppose our registers are 32-bits long. Now, there are 2^32 distinct 32-bit bit patterns. If we interpret registers using natural-number semantics, these 2^32 bit patterns will represent the natural numbers from 0 to 2^32 - 1.

Example: In a 3-bit register, using natural-number semantics, we have the following interpretations:

000 = 0, 001 = 1, 010 = 2, 011 = 3, 100 = 4,

101 = 5, 110 = 6, and 111 = 7.

Hardware can be designed to add, subtract, multiply, and divide the numbers represented by these bit patterns.

How do we represent integers, which may be negative? After a bit of confusion, sensible people decided that integers ("signed numbers" in the jargon) should be represented using _two's complement semantics_. Basically, leading 0s mean positive, while leading 1s mean negative. A slightly larger example is necessary.

<u>Example</u>: In a 4-bit register, using two's complement semantics, we have the following interpretations:

0000 = 0, 0001 = 1, 0010 = 2, 0011 = 3, 0100 = 4,

0101 = 5, 0110 = 6, 0111 = 7, 1000 = -8, 1001 = -7, 1010 = -6, 1011 = -5,

1100 = -4, 1101 = -3, 1110 = -2, and 1111 = -1.

Since 0 occupies a position as a positive number, we wind up with one more nonzero negative number than nonzero positive number. Notice that you only need to look at the most significant bit to decide if a number is positive or negative. This bit is called the _sign bit_.

The addition algorithm is semantics independent. Even so, we may _impose_ a semantics of our choice on all bit patterns to interpret the final result. Let's add a few 4-bit numbers in 4-bit registers and ignore _carry out_.

Then, using two's-complement semantics, we will determine whether we got the right answer in this semantics using 4-bit registers.

```
Example:  1100 = -4       Example:  0100 =  4
         +1100 = -4                +0100 =  4
          ----                      ----
        1|1000 = -8 :-)             1000 = -8 :-(

Example:  1011 = -5       Example:  0101 =  5
         +1011 = -5                +0101 =  5
          ----                      ----
        1|0110 =  6 :-(             1010 = -6 :-(
```

If we discard the most negative integer, we observe that every (two's complement) integer fits into a 4-bit register if and only if its negation fits into a 4-bit register. There is nothing special about 4. The same assertion holds for (two's complement) integers and n-bit registers.

By the way, a positive (two's complement) integer 'x' fits into an n-bit register if and only if x <= 2^(n-1) - 1.

We say that _overflow_ has occurred if the register is too small to contain the correct result, including the sign bit. Note that a _carry_ out of the register is not in itself a sign of overflow. (See first example).

Trick number 1: How to negate a two's complement number.

```
1) Flip every bit.           0101 =  5     0001 =  1
                             1010          1110
2) Add one.                  1011 = -5     1111 = -1

                             1000 = -8
                             0111
                             1000 = -8
```

Ah, not every two's complement number can be negated. Why? Because there is no 4-bit +8. The correct answer must fit in the register.

Trick number 2: How to place an n-bit two's complement number in a register with more than 'n' bits.

1) Copy the number on the right.   Example: 1000 = -8 (4 bits)

2) Replicate the sign bit on the left.   1111 1000 = -8 (8 bits)

Now, let's learn how to write bit patterns in hexadecimal. This is just shorthand: it is _not_ a new semantics.

0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	-8
9	1001	9	-7
a	1010	10	-6
b	1011	11	-5
c	1100	12	-4
d	1101	13	-3
e	1110	14	-2
f	1111	15	-1

We can convert "hex" patterns into bit patterns,

and bit patterns into "hex" patterns.

If the length of the bit pattern is not a multiple

of 4, go from right to left. Pad with zeros.

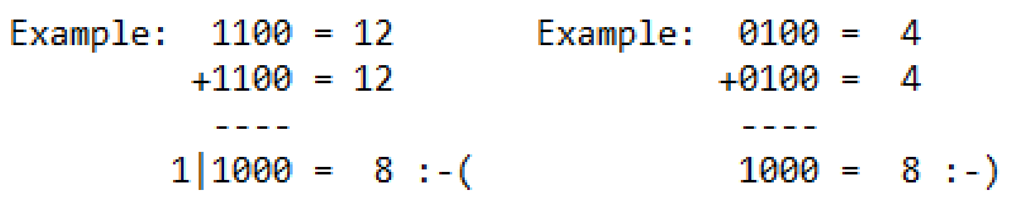
A hex digit corresponds to 4 bits.

A hex digit is a "hexit".

In this table, the columns are:

- i) the hex digit,
- ii) the bit pattern,
- iii) the natural-number semantics
- iv) the two's-complement semantics.

Addendum: It is not the case that two's-complement semantics is somehow the _correct_ semantics. In some contexts, we may prefer natural-number semantics. This leads to an apparent paradox. Redo the first two examples with natural-number semantics.



```
Example:  1100 = 12       Example:  0100 =  4
         +1100 = 12                +0100 =  4
          ----                      ----
        1|1000 =  8 :-(             1000 =  8 :-)
```

What previously was right is now wrong, and vice versa. As it happens, we have no way of telling the adder which semantics we are using. This seems to make error reporting impossible. We will resolve this issue later.

Hexadecimal Integer Manipulation

____

Here is a table:

```
Hex table:                                         Hex flips:      Hex powers:

0    0000    4    0100    8    1000    c    1100    0 - f   4 - b    1, 16, 256, 4096
1    0001    5    0101    9    1001    d    1101    1 - e   5 - a
2    0010    6    0110    a    1010    e    1110    2 - d   6 - 9    Hex naturals:
3    0011    7    0111    b    1011    f    1111    3 - c   7 - 8    a  b  c  d  e  f
                                                                     10 11 12 13 14 15
```

Recall that _no_ integer bit pattern specifies its semantics, natural number [nn] or two's complement [2c], but both the programmer and the architecture can. Semantics helps determine whether a number fits into a register or whether the addition of two numbers is correct. I will give some examples of computing directly in hex below. The register size is indicated by the largest number of hex digits I show. Can you spot the errors in c) below?

```
4 bits:  nn  0..15      2c  -8..7       (1 hex digit)
8 bits:  nn  0..255     2c  -128..127   (2 hex digits)
12 bits: nn  0..4095    2c  -2048..2047 (3 hex digits)

a) dec to hex: 210 ==> <13,2> ==> d2
               -46 ==> <13,2> ==> d2
               105 ==> <6,9>  ==> 69

b) hex negation:  start with a6 (-90)      start with 0a6 (166)
                  flip       59            flip       f59
                  add 1       1            add 1        1
                             --                       ---
                             5a (90)                  f5a (-166)

c) hex addition:  start with 69 (105)      start with  7f (127)
   [in 2c]        add        a6 (-90)      add         fb (-5)
                             --                        --
                           1|0f (15)                 1|7a (122)

                  start with d2 (-46)      start with 0d2 (210)
                  add        a6 (-90)      add        fa6 (-90)
                             --                       ---
                           1|78 (-136)              1|078 (120)

                  start with fd2 (-46)
                  add        fa6 (-90)
                             ---
                           1|f78 (-136)

d) binary expansion: 1/7  ==> 0.(001)*  (infinite decimal expansion)
                     1/14 ==> 0.0(001)*
```