Computer Simulation of a Computer

Charles S. Peskin
Courant Institute of Mathematical Sciences, New York University
May, 2025

In these notes and the accompanying matlab programs, we describe the computer simulation of a computer, at the architectural level. The computer that we simulate has two main parts, a central processing unit (cpu) and a memory.

The memory of our computer is a $2^{13} \times 16$ array to which we give the name `mem`. Each element of `mem` stores one bit. The memory contains the user's application program, and also any data on which that program operates or which that program generates. This dual use of memory for both program and data, with no sharp distinction between the two, is an invention of von Neumann and is known as the von Neumann architecture. It makes it possible for a program to rewrite itself as it runs. Each row of `mem` is regarded as a *word* of memory, so our computer memory contains 2^{13} 16-bit words. Each word of memory has an address, which is simply the index of its row in the array `mem`. These indices conceptually run from $0 \ldots 2^{13} - 1$, so that they can be represented by 13-bit binary numbers, but matlab does not allow an array index to be zero, so we add 1 to the conceptual address to get the corresponding array index. Thus, in matlab, `mem(1+addr,:)` denotes the array of bits corresponding to the memory word with the address `addr`.

A word of memory may contain an instruction that needs to be executed by the cpu. In that case, the first 3 bits of the word are the instruction code, and the remaining 13 bits encode as a binary number the address in memory to which the instruction refers. What is done with or to the data at that address in memory depends on the instruction, as described below. Since the instruction code contains 3 bits, there are $2^3 = 8$ instructions that our cpu can carry out.

In general, if the address part of an instruction is `addr`, then depending on the instruction the cpu may read from or write to the memory word with address `addr`, and it may also read from or write to a 16-bit array within the cpu that is called the register, and denoted `reg`. The cpu that we describe here has only a single register, and this makes it more challenging to write application programs for our computer than if more registers were available, but the lack of additional registers does not fundamentally limit what our computer can do.

The instruction set of our simulated computer is as follows:

- 000 LOAD: copy the content of `mem(addr)` into `reg`
- 001 STORE: copy the content of `reg` into `mem(addr)`
- 010 ADD: add the content of `mem(addr)` to the content of `reg`, treating both as 16-bit binary numbers, and store the result (mod 2^{16}) in `reg`
- 011 BNZ (branch if nonzero): if any of the 16 bits of `reg` is nonzero, reset the program counter so that the next instruction to be executed is the instruction contained in `mem(addr)`
- 100 AND: perform bitwise AND of the content of `reg` with the content of `mem(addr)`, and store the result in `reg`
- 101 OR: perform bitwise OR of the content of `reg` with the content of `mem(addr)`, and store the result in `reg`
- 110 XOR: perform bitwise exclusive-OR of the content of `reg` with the content of `mem{addr)` and store the result in `reg`
- 111 NOT: complement every bit of `reg`

The only instruction that actually changes the content of the memory is STORE. When a STORE operation is performed, whatever was in the affected memory location previously is erased and replaced by the content of the register (which remains unchanged). All of the operations other than STORE and BNZ put their results into the register, and whatever was in the register previously is erased (and the content of memory is unchanged). The only conditional operation is BNZ. Depending on the content of the register, it either does nothing or resets the program counter, which is a variable located within the cpu that determines which word of memory will be executed next (more about this below). The only instruction that does not refer to memory at all is NOT, which complements each bit of the register.

We now turn to a description of the program that is run by the cpu. Note that this is always the same program, regardless of the user's application program. In practice the program of the cpu is implemented in hardware , but we can simulate its logic in a programming language such as matlab:

```
%cpu_program.m
%mem is the central memory,
%an array of bits with 2^13 rows and 16 columns.
%Conceptually, the rows of mem are numbered 0...((2^13)-1),
%but we have to add 1 in matlab.  This will be written "1+addr".

%When a row of mem is an instruction,
%the first three bits are the operation code,
%and the last 13 bits are the associated address,
%that is, the address on which the instruction operates
%by reading or writing there.

reg=zeros(1,16); pc=1; %initialize register and program counter
while pc > 0           %setting pc to zero will stop the program
  ins= num(mem(1+pc,1:3),3);   %read instruction code
  addr=num(mem(1+pc,4:16),13);  %read address
  %increment program counter to prepare for next step:
  pc=mod(pc+1,2^13);
  switch ins %execute instruction specified by ins
    case 0  %LOAD
      reg=mem(1+addr,:);  %copy mem line addr into register
    case 1  %STORE
      mem(1+addr,:)=reg;  %copy register into mem line addr
    case 2  %ADD mod 2^16
      %convert bits in register and mem line addr to integer:
      s1=num(reg,16);
      s2=num(mem(1+addr,:),16);
      %add and store resulting bits in register:
      reg=bin(s1+s2,16);
    case 3  %BRANCH on NONZERO REG
      if(any(reg)) %if any bit of register is nonzero,
        pc=addr; %set program counter to addr for next step
      end
    %In the following, all results are stored in register;
    %memory is read (except NOT), but not changed:
    case 4 %AND
      %bitwise AND of reg with mem line addr:
      reg = reg & mem(1+addr,:);
```

```
    case 5 %OR
      %bitwise OR of reg with mem line addr:
      reg = reg | mem(1+addr,:);
    case 6 %XOR
      %bitwise exclusive OR of reg with mem line addr:
      reg = xor(reg,mem(1+addr,:));
    case 7 %NOT
      %bitwise NOT of register (addr is ignored):
      reg = ~reg;
  end
end
```

The above program calls two functions `num` and `bin`, listed below. The function `num` interprets a vector of bits as a binary number, and evaluates that binary number as an integer. The function `bin` finds the binary representation of an integer, and outputs that representation as a vector of bits.

```
function n=num(b,nbits)
% b = vector of length nbits with entries equal to 0 or 1
% n = integer equivalent of the binary number
% with bits b(1)...b(nbits)
powers=nbits-(1:nbits);
n=sum(b.*(2.^powers));

function b=bin(n,nbits)
%n is a non-negative integer
%b = vector of length nbits with entries 0 or 1 such that
%b is the binary representation of the n mod 2^nbits

%This is a recursive program -- it calls itself!

if nbits==1
  b=mod(n,2);
else
  nn=floor(n/2);
  b=[bin(nn,nbits-1),n-2*nn];
end
```

The function num is self-explanatory, but bin is a recursive program that may not be easy to understand. The function bin is based on two facts. First, the least significant bit in the binary representation of n is n-2*floor(n/2). Also, the part of the binary number to the left of the least significant bit is the binary representation of floor(n/2).

The program cpu_program.m that is executed by the cpu begins by setting all bits of the register reg equal to zero and by setting the program counter pc equal to 1. The rest of the program is in a while loop over pc > 0, so the program will stop when pc becomes zero. After each instruction is executed, pc is incremented mod 2^{13} by 1, so one way the program will stop is if pc wraps around to zero. But pc can also be set to any value by execution of a BNZ instruction when at least one bit of the register is not equal to zero. Thus, a two-instruction sequence that is guaranteed to stop the program is:

```
LOAD ONE
BNZ ZERO
```

Here ONE is the address of a word of memory that contains the value 1, and ZERO is the memory location whose address is zero. (By convention, we will also keep the value zero in the memory location whose address is zero, but that is not relevant here.)

Within the while loop of cpu_program.m, the first step is to read the instruction to which pc is pointing, and to separate that instruction into its 3-bit instruction code and its 13-bit address. These binary vectors are converted to the integers that they represent by calls to the function num. The next step is to increment the program counter. This is done sooner rather than later because the program counter might be re-set by a BNZ operation, and we do not want the increment of the program counter to overwrite a change in pc that might be made if the instruction to be executed is BNZ. The rest of the while loop is a switch statement that executes the specified instruction with reference to the specified address.

Note that initialization of mem is not done in cpu_program.m. It is the responsibility of the user to put the application program along with any data that it needs into memory before calling cpu_program.m

We give an example of such an application program here:

```
%multiply_binary.m
%set up and run machine language program to compute
%   P = A * B mod 2^16
mem=zeros(2^13,16);

%instruction codes:
LOAD  = bin(0,3);
STORE = bin(1,3);
ADD   = bin(2,3);
BNZ   = bin(3,3);
AND   = bin(4,3);
OR    = bin(5,3);
XOR   = bin(6,3);
NOT   = bin(7,3);

%Line numbers in mem at which constants and variables will be stored:
%Note that these are the *line numbers*, not the values.
%Values will be assigned later.
%Line numbers chosen big enough to be out of the way.

BC = 99;
ONE = 100;
BIT = 101;
A = 102;
B = 103;
P = 104;

%Names for some line numbers in the program:
ZERO = 0;   %ZERO is both a line number and a constant (see below)
BACK = 8;
CONT = 14;

%Write the program:
mem(1+ZERO,:)=zeros(1,16);          %branch here to stop
mem(1+1,:)=[LOAD,bin(ZERO,13)];     %program starts here
mem(1+2,:)=[STORE,bin(P,13)];       %initialize P=0
```

```
mem(1+3,:)=[LOAD,bin(ONE,13)];
mem(1+4,:)=[STORE,bin(BIT,13)];    %initialize BIT=1
mem(1+5,:)=[LOAD,bin(B,13)];
mem(1+6,:)=[NOT,bin(ZERO,13)];
mem(1+7,:)=[STORE,bin(BC,13)];     %initialize BC = NOT B
mem(1+BACK,:)=[LOAD,bin(BC,13)];   %main loop starts here
mem(1+9,:)=[AND,bin(BIT,13)];
mem(1+10,:)=[BNZ,bin(CONT,13)];    %if relevant_bit(B) is 1  ...
mem(1+11,:)=[LOAD,bin(P,13)];
mem(1+12,:)=[ADD,bin(A,13)];
mem(1+13,:)=[STORE,bin(P,13)];     %P=P+shifted A
mem(1+CONT,:)=[LOAD,bin(BIT,13)];  %CONTINUE here in either case
mem(1+15,:)=[ADD,bin(BIT,13)];
mem(1+16,:)=[STORE,bin(BIT,13)];   %BIT=left_shift(BIT)
mem(1+17,:)=[LOAD,bin(A,13)];
mem(1+18,:)=[ADD,bin(A,13)];
mem(1+19,:)=[STORE,bin(A,13)];     %A=left_shift(A)
mem(1+20,:)=[BNZ,bin(BACK,13)];    %if A is nonzero, goto BACK
mem(1+21,:)=[LOAD,bin(ONE,13)];    %else ...
mem(1+22,:)=[BNZ,bin(ZERO,13)];    %STOP

%Assign value to the constant ONE:
mem(1+ONE,:)=bin(1,16);

%ZERO is both a line number and a constant.
%The value 0 was already assigned when the program was written.

%Assign values to the variables that will be multiplied
Avalue=input('Avalue=');
Bvalue=input('Bvalue=');
mem(1+A,:)=bin(Avalue,16);
mem(1+B,:)=bin(Bvalue,16);

%run the program:
cpu_program

%output the result:
Pvalue = num(mem(1+P,:),16)
```

The above program is written in an informal assembly language that is translated to machine language within the program itself, so a separate assembler is not needed. In assembly language, names are used instead of numbers, so that the program becomes somewhat readable. There is a name for each instruction in the instruction set, and also a name for each memory location where a variable will be stored, and finally there are names for some line numbers that need to be referenced within the application program. The task of the assembler is to assign numbers to these names, so that the whole program can be reduced to an array of bits. Here we make the assignments by hand.

The algorithm that we use for binary multiplication is essentially the same as the one that is taught in elementary school for decimal multiplication. The binary case is simpler, however, since we only need to multiply by 0 or 1. The following example will suffice to recall the algorithm:

```
                1   1   0   1
          x         1   0   1
                ---------------
                1   1   0   1
            0   0   0   0
        1   1   0   1
-------------------------------
1   0   0   0   0   0   1
```

In the above method, successive left-shifts of the first factor are considered, and those corresponding to the locations of 1 in the second factor are kept and added together. Our computer does not have a left-shift operation, but left-shift in binary is equivalent to multiplication by 2, and a number can be multiplied by 2 by adding it to itself!

The above program `multiply_binary.m` implements this algorithm in the following way. We give here the pure assembly language version, which should be easier to follow, but which requires an assembler to be of any practical use:

```
      LOAD  ZERO
      STORE P       % P = 0
      LOAD  ONE
      STORE BIT     % BIT = 1
      LOAD  B
      NOT
      STORE BC      % BC = NOT B
BACK  LOAD  BC      % main loop starts here
      AND   BIT     % look only at one bit of BC
      BNZ   CONT    % if nonzero skip update of P
      LOAD  P       % else ...
      ADD   A
      STORE P       % P = P + shifted A
CONT  LOAD  BIT
      ADD   BIT
      STORE BIT     % BIT = left_shift(BIT)
      LOAD  A
      ADD   A
      STORE A       % A = left_shift(A)
      BNZ   BACK    % if A is nonzero, repeat with shifted data
      LOAD  ONE     % else...
      BNZ   ZERO    % stop; answer is in P
```

The numbers to be multiplied are assumed to be initially stored in the memory location named `A` and `B`. After the program stops, their product will be stored in the memory location named `P`. Line number 0 of memory has been given the name `ZERO`, and this line of memory also contains the binary representation of 0. (This is a special case – it is not typical for the address of a word of memory to be the same as the content of that word. Indeed the address is a 13-bit binary number, whereas the content is a 16-bit binary number.) There is also a line (i.e., word) of memory to which we have given the name `ONE`, and it contains the binary representation of 1. It is useful to have the complement of the content of `B` available, and this is stored in the line number to which we have given the name `BC`. The contents of `B` and `BC` do not change as the algorithm proceeds.

There is a memory line with the name BIT that is initialized to contain the binary representaion of 1, and its one nonzero bit is left-shifted after each step of the algorithm. This is accompanied by a left-shift of the binary represention of the content of the memory line whose name is A.

At each step of the algorithm, the content of the memory line with the name BIT and the content of the memory line with the name BC are combined by a bitwise AND operation, and the result has all bits equal to zero except possibly the one bit in the content of BC in the same position as the one nonzero bit of the content ofBIT. This one bit is then used to decide whether to update the product by addition of a shifted version of the first factor (if the bit is 0) or to skip the update (if the bit is 1). (Recall here that the bitstring in BC is the complement of the bitstring in B.)

If the function of the program is not yet clear, it may be good to carry out its steps by hand on some small example.

Binary addition

The computer we have been considering up to now has an ADD operation, but no left-shift operation, so to generate a left-shift we had to add a number to itself. We consider now the opposite situation, in which the computer is the same as before except that it does not have an ADD operation, and it instead has a left-shift operation, denoted LSHIFT. Like the operation NOT, the LSHIFT operation is applied to the register and makes no reference at all to central memory. Its effect is to discard the left-most bit of the register, and to move all of the other bits one step to the left, with the right-most bit of the register being set equal to zero. In matlab-like notation, this is the operation

```
reg = [reg(2:16),0]
```

In this computer, we need a program to add two 16-bit binary numbers, each of which is stored in a word of memory. The addition is to be done modulo 2^{16}. The strategy that we shall use is based on the following considerations, in which all all arithmetic is understood to be modulo 2^n with $n = 16$:

Let

$$A = \sum_{k=1}^{n} a_k 2^{n-k}, \quad a_k \in \{0,1\} \tag{1}$$

$$B = \sum_{k=1}^{n} b_k 2^{n-k}, \quad b_k \in \{0,1\} \tag{2}$$

Then

$$A + B = \sum_{k=1}^{n} (a_k + b_k)\, 2^{n-k} \tag{3}$$

The expression $(a_k + b_k)$ can also be written in terms of logical operations in the following way:

$$a_k + b_k = (a_k \ \texttt{XOR} \ b_k) + 2\,(a_k \ \texttt{AND} \ b_k) \tag{4}$$

in which $1 =$ `TRUE` and $0 =$ `FALSE` . Equation (4) is easily checked by considering all four possible values of the pair (a_k, b_k). It follows that

$$A + B = A' + B' \tag{5}$$

where

$$A' = \sum_{k=1}^{n} (a_k \ \texttt{XOR} \ b_k)\, 2^{n-k} \tag{6}$$

$$B' = 2\sum_{k=1}^{n} (a_k \ \texttt{AND} \ b_k)\, 2^{n-k} \tag{7}$$

Here A' is the result of doing binary addition without any carrying, and B' contains all of the carry bits. but the carry bits have not yet been carried to the place in which they are needed — the operation `LSHIFT` is needed for that, see below.

Note that A' already has the form of an n-bit binary number. In the formula for B' we may discard the term $k = 1$ because it is either 0 or $2\,(2^{n-1}) = 2^n = 0$ modulo 2^n. Therefore,

$$\begin{aligned} B' &= \sum_{k=2}^{n} (a_k \ \texttt{AND} \ b_k)\, 2^{n+1-k} \\ &= \sum_{k=1}^{n-1} (a_{k+1} \ \texttt{AND} \ b_{k+1})\, 2^{n-k} \end{aligned} \tag{8}$$

Thus B' also has the form of an n-bit binary nmber, and moreover the least significant bit ($k = n$) of B' is 0.

The formulae for the bits of A' and B' in terms of the bits of A and B are as follows:

$$\begin{aligned} a'_k &= (a_k \ \texttt{XOR} \ b_k), \ \ k = 1, \ldots, n && (9) \\ b'_k &= (a_{k+1} \ \texttt{AND} \ b_{k+1}), \ \ k = 1, \ldots, n-1 && (10) \\ b'_n &= 0 && (11) \end{aligned}$$

The above results motivate the following iterative scheme for binary addition:

$$a_k^{(0)} = a_k, \ \ b_k^{(0)} = b_k, \tag{12}$$

and for $m = 0, 1, 2, \ldots$,

$$\begin{aligned} a_k^{(m+1)} &= a_k^{(m)} \ \texttt{XOR} \ b_k^{(m)}, \ \ k = 1, \ldots, n && (13) \\ b_k^{(m+1)} &= a_{k+1}^{(m)} \ \texttt{AND} \ b_{k+1}^{(m)}, \ \ \ldots, n-1 && (14) \\ b_n^{(m+1)} &= 0 && (15) \end{aligned}$$

In more succinct notation, the above scheme reads as follows:

$$A^{(0)} = A, \ \ B^{(0)} = B, \tag{16}$$

and for $m = 0, 1, 2, \ldots$,

$$\begin{aligned} A^{(m+1)} &= A^{(m)} \ \texttt{XOR} \ B^{(m)} && (17) \\ B^{(m+1)}) &= \ \texttt{LSHIFT} \ \left(A^{(m)} \ \texttt{AND} \ B^{(m)}\right) && (18) \end{aligned}$$

This scheme has the property that $A^{(m)} + B^{(m)} = A + B$ for every m.

Now it may not seem that we have accomplished much, since we have just replaced a problem of binary addition by a sequence of other problems of the same form. Note, however, that at each stage of the above algorithm, a 0 is introduced in the n^{th} bit of B. As these zeros propagate to the left because of the left-shift operation, they remain zero because $a \ \texttt{AND} \ 0 = 0$. Thus, after at most n steps, all of the bits of $B^{(m)}$ are zero, and $A^{(m)}$ contains the sum $A + B$.

The assembly-language implementation of the above algorithm is very simple:

```
ZERO
BACK   LOAD    A
       XOR     B
       STORE   A        %  A = A XOR B
       XOR     B        %  recover previous A in register
       AND     B
       LSHIFT
       STORE   B        %  B = LSHIFT(A_previous AND B)
       BNZ     BACK     %  if not done, repeat
       LOAD    ONE      %  else
       BNZ     ZERO     %  STOP
```

The reversible nature of `XOR` is exploited in the foregoing to recover the previous value of `A` in the register after what is stored in the register has been changed from `A` to `A XOR B`. The second `XOR B` operation restores the content of the register to what it was right after the `LOAD A` statement. This avoids the need for an extra variable to hold a copy of `A`.

A possible project is to implement this program in the same manner as `multibly binary.m`. Note the the program cpu.m then has to be modified to omit the `ADD` operation and replace it by `LSHIFT`.

Arrays, loops, and a program that rewrites itself

In this section we consider the implementation of arrays and loops in the context of the simulated computer introduced above (as originally defined with `ADD` as one of its operations).

An array is simply a collection of adjacent memory locations to which we assign a base address and a name. If the base address is `base_addr` and the name of the array is `A`, then the array element `A(i)` is stored in the memory location with address `base_addr + i`. In a typical use of this setup, the index `i` takes values in $\{1, \ldots, n\}$ for some positive integer n, but there in nothing to prevent `i` from being zero or even negative. It is the responsibility of the programmer to prevent the array index from going outside of its intended domain.

Although we are only considering one-dimensional arrays here, it should be noted that multidimensional arrays are typically implmented as one-dimensional

arrays, with conversion internally from a multi-index to the corresponding single index.

Instructions that refer to an array element can be handled in the following way. Suppose, for example, we know that a program needs an instruction which conceptually is `LOAD A(i)`. We can allocate a memory location to which we give the name `LOADA`. The contents of this word of memory, which remain constant during program execution, are the 3-bit instruction code for `LOAD` followed by the 13-bit base address of the array `A`. Then, in a program, at a place where `i` has been defined and where the instruction `LOAD A(i)` will soon be needed, we can execute the following instructions:

```
LOAD  LOADA
ADD   i
STORE ILA
```

Here `ILA` is the name of the line of memory where the intruction `LOAD A(i)` is needed. An example of a program that uses this technique will be given below.

Programs that include arrays generally involve loops over the array index. In the context of our virtual computer, this is most easily done by letting the index run down in steps of -1, so that the loop is complete when the index reaches zero. The reason for this is that the only conditional operation available to us is `BNZ`, which resets the program counter to a specified location in memory when the contents of the register is not zero. Thus, a typical loop will have the following form

```
      LOAD n
BACK  STORE i
      % insert here code for instance i of the loop
      LOAD i
      ADD DECR
      BNZ BACK
      % code continues from here when loop is complete
```

The constant `DECR` in the foregoing has all of its 16 bits equal to 1. This is the binary representation of $2^{16} - 1$, which is equivalent to -1 in the arithmetic done by the `ADD` operaction. Thus `ADD DECR` has the effect of subtracting 1 from whatever is in the register (and storing the result in the register).

Now we are ready to illustrate the use of the above techniques in an application program. Given an array of nonnegative numbers $A(1), \ldots, A(n)$, we seek an

index i such that $A(i) \leq A(j)$ for $j = 1, \ldots, n$. Of course there may be more than one value of i with this property, and if so we just want one of them.

The algorithm that we will use to find such an i is far from being efficient, but it is very simple and can be used to illustrate the above programming techniques without a lot of irrelevant complication. The strategy is simply to reduce all of the array elements by 1, and to do this repeatedly until one of them hits zero. (An important detail is that the check for zero needs to be done *before* the reduction by 1, so if there is already a zero in the given array data, it will not be missed. The assembly language program is as follows:

```
ZERO                  %branching here stops the program
BACK1  LOAD n
       STORE i
BACK2  LOAD LOADA
       ADD i          % create the instruction LOAD A(i),
       STORE ILA      % and store it in ILA.
       LOAD STOREA
       ADD i          % create the instruction STORE A(i),
       STORE ISA      % and store it in ISA.
ILA                   % LOAD A(i)
       BNZ CONT       % if A(i) is nonzero, continue
       LOAD ONE       % else
       BNZ ZERO       % STOP (result is in i)
CONT   ADD DECR       % A(i) := A(i) - 1  (in register)
ISA                   % STORE A(i)
       LOAD i
       ADD DECR       % i := i - 1  (in register)
       STORE i
       BNZ BACK2      % consider next element, if any
       LOAD ONE       % else
       BNZ BACK1      % start next sweep through array
```

The above program has two nested loops. The inner loop sweeps through the array once, decreasing each element by one. The outer loop is does as many sweeps as may be needed for one of the elements of the array to reach zero. The program stops by breaking out of the inner loop when a zero element is found, and the index of that element is then to be found in the memory location whose name is `i`. The original value of `A(i)` is of course lost during the execution of the

algorithm, so if it is wanted, a copy of the array should be made before executing the above code, or alternatively, a count can be made of the number of sweeps that were required to reduce `A(i)` to zero in steps of -1.

Project suggestions

In order of increasing difficulty, with the last one much more difficult than the others:

- implement either of the programs in the previous two sections in the same style as `multiply_binary.m`
- think of another application that can be done by our virtual computer, write an assembly language program for that application, and implement it, again following the style of `multiply_binary.m`
- formalize the assembly language that has been used in the foregoing, and write an assembler/loader that will automate the steps of translation to machine language, loading the program into memory, and execution of the program (by calling `cpu_program.m`)

Acknowledgement

Thanks to Eric Peskin for helpful and enjoyable discussions related to these Notes!