CPG 5 Coding and Programming

Introduction to Game Programming

Paul Sinnett, London South Bank
University <paul.sinnett@gmail.com>

Table of Contents

Week 6

Machine code

This week's subject areas are:

- the programmable logic chip (ALU)
- the program counter (PC) register
- the central processing unit (CPU)
- a complete machine code

The programmable logic chip (ALU)

Last week we looked at building game rules into hardware. We took some rules from a game design and rewrote them as combinations of a minimal set of logical operations. The same minimal set of operations that we've built up in earlier classes. The operations were:

- NOT
- AND
- OR
- ADD
- NEGATE
- == 0?
- < 0?

With these rewritten rules, we could design circuits to apply the rules, usually in a continuous loop. With the rules of a game constructed in this way it should be possible to completely realize any game design.

But constructing games in this way has many drawbacks:

- laborious
- error prone
- expensive

- difficult to modify

What we really need is a more general system which can read our rules somehow and process them as if we'd built them into hardware. That way if we want to add more rules, or change the rules, we only have to get the device to re-read the rules and we'd have a new game without having to add any new bits or rewire any circuits.

Figure 1. The Jacquard Loom

Joseph Marie Jacquard was a weaver and inventor. He created a programmable loom in the early 1800's. The loom was programmed with punched cards; literally cards with holes punched into them. The holes encoded instructions to the loom to raise and lower certain stands. This allowed the loom to semi-automatically weave in the patterns that were encoded on the cards.

Figure 2. Bitmap data encoded on punched cards

Jacquard's punched card system allowed the machines to be given any pattern and weave it into fabric in much the same way that we create and draw bitmaps on computers today. Charles Babbage owned a portrait of Jacquard woven in silk by a loom programmed with over 20,000 punched cards. This inspired Babbage to design a similar programming system for his analytical engine.

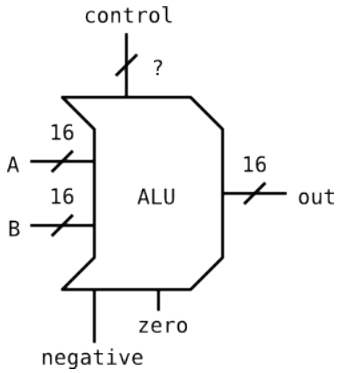
We've already looked at how we can program the address inputs of a memory chip to allow us to read or write to different registers inside the memory. Next we're going to extend that idea to allow us to access different logical circuits inside a programmable logical chip.

We could simply follow the memory chip design, but with the address selecting between different logical circuits instead of different registers. This is a straight forward approach, but it would introduce a lot of redundancy since many of the operations we want to perform are simply variations of the same circuit.

Instead we'll attempt to encode a minimal set of operations into a single chip design and then test to see if it can be programmed to perform any of the standard operations we want to perform on two input values. The black box of this is that we want a circuit that takes 2 inputs, and outputs the result of applying some operation, selectable by some number of control bits. It also outputs a couple of indicator bits, one if the result is zero, another if the result is negative.

Figure 3. Arithmetic logic unit

A black box view of the ALU circuit. It takes two 16 bit inputs, and outputs the 16 bit result of applying some operation. The operation is selectable by some number of control bits. It also outputs a couple of Boolean indicator bits, one if the result is zero, another if the result is negative.



I'm going to break this down into four stages that I'm going to call: filter, pre-flip, apply, and post-flip. The first two stages apply to both inputs, the third stage combines the inputs, and the forth stage operates only on the output.

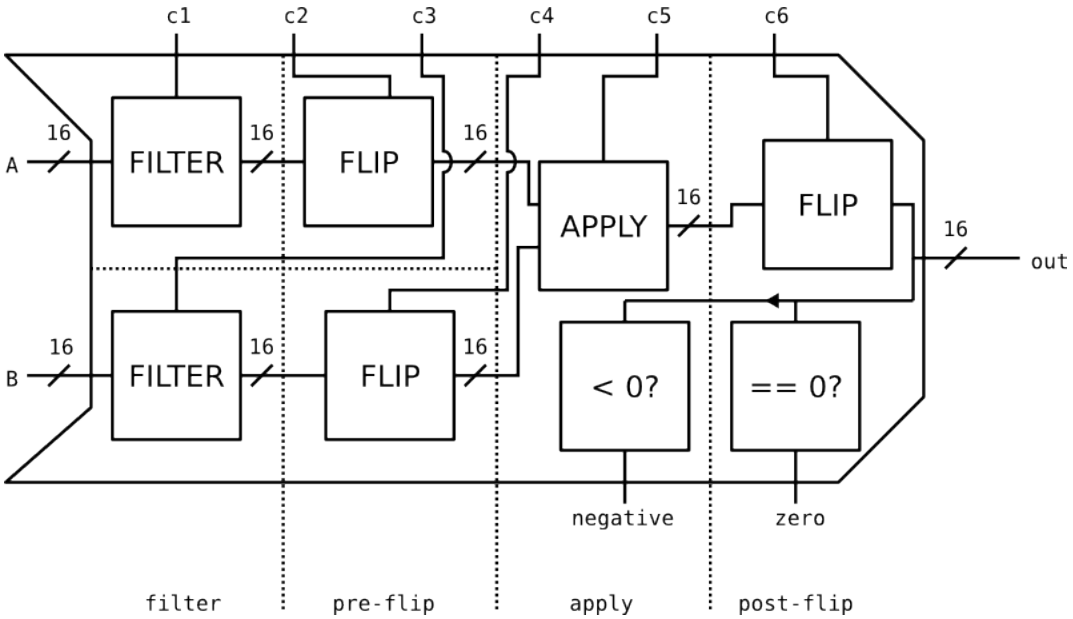
Figure 4. ALU design

One way to implement the above ALU. This circuit requires the implementation of 3 more black box circuits which I'm going to call: FILTER, FLIP, and APPLY.

The FILTER circuit takes a 16 bit input and a control bit. If the control bit is 0, the output is the same as the input. If the control bit is 1, the output is 0. We can implement this by using a selector (16 bit DMUX) to pick between the input and 0.

The FLIP circuit takes a 16 bit input and a control bit. If the control is 0, the output is the same as the input. If the control bit is 1, each bit in the output is the opposite of the input. We can implement this by using a selector to pick between the input as it is and the input running through a 16 bit NOT gate.

The APPLY circuit takes two 16 bit inputs and a control bit. If the control bit is 0, the output is the bitwise AND of the inputs. If the control bit is 1, the output is the sum of the inputs. We can implement this by splitting the inputs, producing both outputs, and then using a selector to pick between them.



Using this arrangement we need only 6 bits to control all of the operations giving us a total of 64 output combinations.

Rather than enumerate and calculate all 64 combinations, let's take a small set of operations we'd like to support and see if and how they can be programmed using this circuit's control bits:

Table 1. Combinations of control bits in ALU

A input		B input		output		Result
c1	c2	c3	c4	c5	c6	
Filter?	Flip?	Filter?	Flip?	ADD?	Flip?	
						`0`
						`1`
						`-1`
						`A`
						`B`
						`~A`
						`~B`
						`-A`
						`-B`
						`A + 1`
						`B + 1`
						`A - 1`
						`B - 1`
						`A + B`
						`A - B`
						`B - A`
						`A & B`
						`A | B`

Combinations of control bits in ALU

Can you fill in the above table to get the required answers?

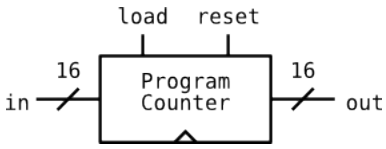
The program counter (PC) register

In the previous section we saw how to build a single chip that we can program to carry out one of a range of calculations on two input values. The programming of this chip requires just 6 bits so we could store the complete instruction for this chip in one 16 bit register and still have 10 bits left to use for other things.

To fully automate the process of programming this chip, we need some way to load our program instructions. We also need to keep track of where we are in a sequence of instructions. Finally we need some way of advancing to the next instruction. All of these abilities are provided by a special circuit known as the program counter.

Figure 5. The program counter

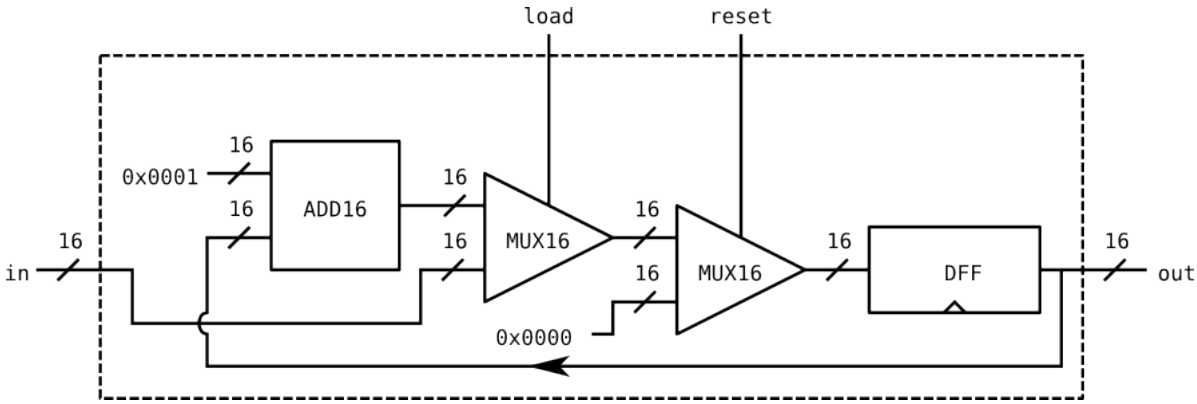
The program counter circuit is a variation of the standard register. It takes a 16 bit input representing an instruction address to load. It has 2 control bits: reset and load. And it outputs a 16 bit value representing the address of the next instruction. If the reset input is set, the circuit outputs 0 as the address value at the next clock. If the load input is set, it outputs the input value as the address at the next clock. Otherwise, it outputs the current value + 1 at the next clock.



It should be simple to implement this circuit using bits and pieces we've already covered. Can you see how?

Figure 6. Building a program counter

Implementing a PC circuit is a simple matter of linking a standard DFF and connecting the input through 2 selectors in a row. The load selector picks between the current value + 1 and the input value to the circuit. The reset selector picks between that and 0.



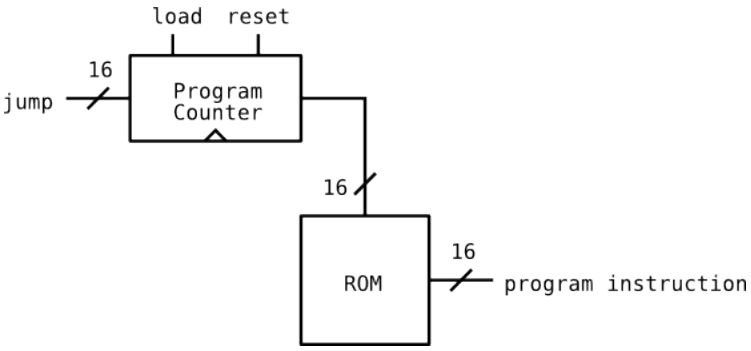
Connecting the output of this circuit to the address input of a RAM or ROM will cause those circuits to emit a sequence of instructions so long as the load and reset inputs are not set. If the load input is set, the next instruction will instead jump to the location encoded in the input to the PC. And if the reset input is set, the next instruction will be the instruction at address 0.

Figure 7. An instruction sequence

So long as the load and reset input are not set, the PC will increase by one at each clock cycle. And the ROM chip will therefore emit the next program instruction in the sequence.

If the load bit is set, the PC will jump to the given jump address at the next clock cycle. This will cause the ROM chip to emit the program instruction at the jump address. Once the load bit is clear, the PC will resume instructions from that point.

If the reset bit is set, the PC will jump to zero at the next clock cycle. This will cause the ROM chip to emit the program instruction at address 0. This effectively starts the program from the beginning again.



We should now have enough to connect together a CPU.

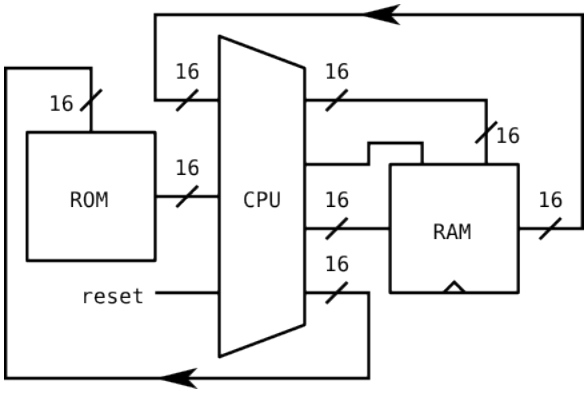
Do you see how this could work?

The central processing unit (CPU)

As we have a programmable circuit that can perform calculations, and another that can emit a sequence of program instructions we should be able to connect them together and make a general purpose programmable circuit. Given enough instructions (ROM) and storage space (RAM), this circuit should be capable emulating any circuit we could imagine.

Figure 8. A simple general purpose computer

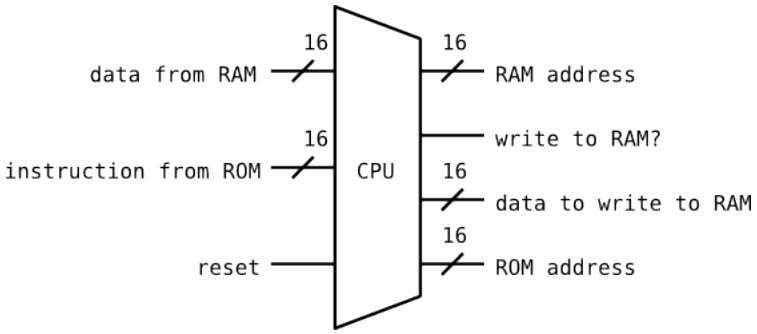
The ROM chip emits instructions which program the CPU to perform an operation. This operation might takes the contents of a register from inside the RAM circuit. On the next clock cycle, it might write out results to a register in the RAM circuit, and advance to either the next instruction, or jump to some other instruction depending on the results of its calculations:



Building such a device requires a circuit that can communicate with a ROM circuit to get its instructions and a RAM circuit to store its current state.

Figure 9. CPU Black Box

The CPU needs 16 bit inputs from the ROM circuit for instruction, and from the RAM circuit for data. It needs to output addresses for both its current place in ROM and RAM. It also needs to output a signal if it wants to write to RAM, and the new data value to place there. Finally, it needs a reset bit to start it at the first instruction.



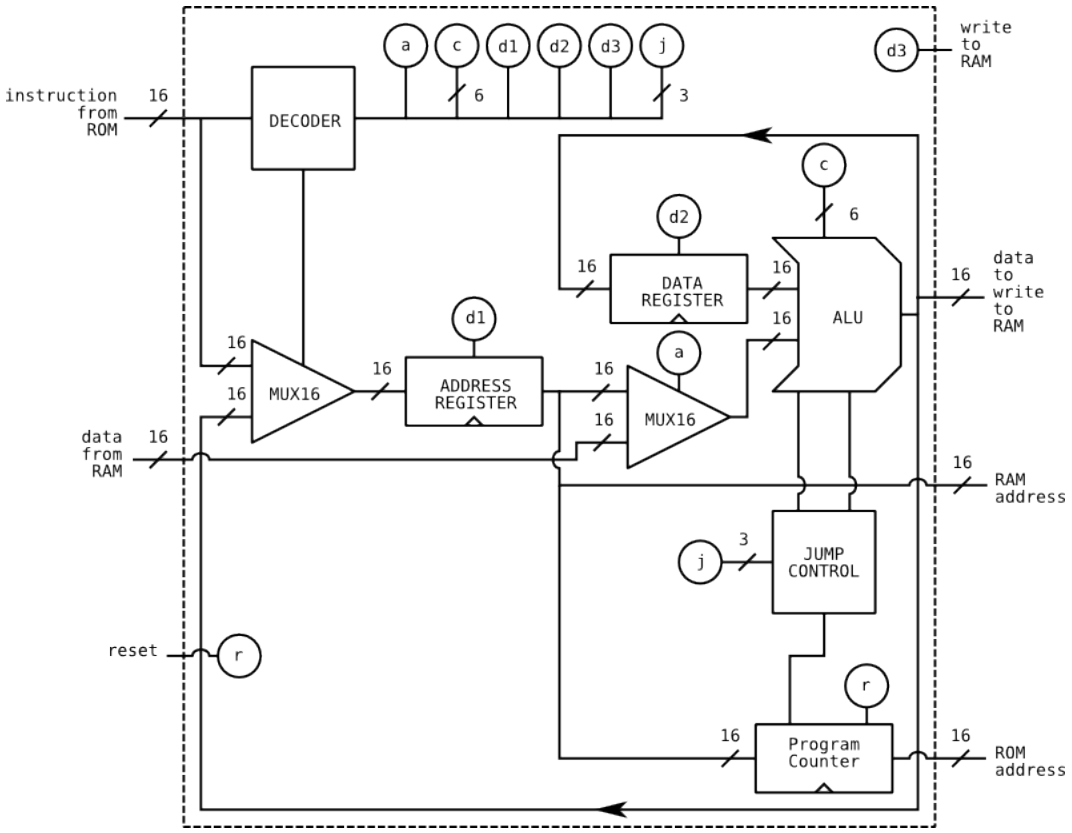
The design of a CPU circuit that will fit this description will need the following components: an ALU, a PC, an address register. Also, since the ALU requires two 16 bit inputs, and the CPU can only receive one 16 bit input from RAM per clock cycle, it will also need a second register for that data. It also needs some way to decode its instructions and pass the appropriate bits to the ALU, and some logic to make the PC jump or increment depending on the output from the ALU.

Figure 10. A simple CPU design

This CPU design needs a couple of additional black box components: a decoder, and a jump controller.

The decoder, takes a 16 bit instruction as input. If the instruction is to load a value it outputs a signal to the address register to load (d1) and to a selector to pick the instruction as the input value. Otherwise it outputs signals to program the ALU, the jump controller, and the various register load inputs. The last of the register load signals is the signal out to the RAM circuit.

The jump controller takes the 3 jump control bits (j) from the decoder and 2 inputs from the ALU (the negative and zero output lines). It outputs a signal to the PC to switch between taking its next value from the address register or continuing with the next instruction.



The jump control bits determine the condition required to cause a jump. The control input bits j1, j2, and j3 are set according to the following table:

Table 2. Jump control bits

condition			**meaning**
j1	**j2**	**j3**	
< 0	**== 0**	**> 0**	
0	0	0	don't jump
0	0	1	if ALU output > 0 jump
0	1	0	if ALU output == 0 jump
0	1	1	if ALU output >= 0 jump
1	0	0	if ALU output < 0 jump
1	0	1	if ALU output != 0 jump
1	1	0	if ALU output <= 0 jump
1	1	1	always jump

Jump control bits

Can you see how to implement the JUMP CONTROL circuit?

Machine code

The placement of the bits within the instruction codes is contained within the decoder. We could arrange these however we like. The following tables describe the layout of the machine code for this CPU design.

There are two main types of instruction: load instructions and compute instructions.

Load instructions load the value of the instruction into the address register of the CPU. If the top bit of the instruction is not set, then the instruction is a load instruction. The remainder of the bits can be set to any number. This number will then be copied into the address register on the next clock:

Table 3. The load instruction

hex digit	**4**				**3**				**2**				**1**			
binary digit	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
value	0	a15	a14	a13	a12	a11	a10	a09	a08	a07	a06	a05	a04	a03	a02	a01

The load instruction

If the top bit is set, the instruction is a compute instruction. The remainder of the bits program various parts of the CPU as follows:

- a - if set, the value from RAM is passed to the ALU
- c1 to c6 - these program the ALU as in the table above
- d1 - if set, loads the output of the ALU into the address register
- d2 - if set, loads the output of the ALU into the data register
- d3 - if set, loads the output of the ALU into RAM
- j1 to j3 - program the jump control as in the table above

Table 4. The compute instruction

hex digit	**4**				**3**				**2**				**1**			
binary digit	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
value	1	1	1	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3

The compute instruction

To write a machine code program for this CPU, all I need to do is work out which bits I need to set to get the CPU to perform the instructions I want and convert them into hexadecimal. Here again is the table to convert between binary digits and hexadecimal digits:

Table 5. Conversion between hexadecimal and binary

Hexadecimal	**Binary**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Conversion between hexadecimal and binary

To begin with I need to define where in RAM I'm going to store the current state of my game. It doesn't matter where in RAM I decide to keep them so long as I always reference the same address throughout my program. I'm just going to start at address 0 and work down:

```
RAM   : data
0x0000: ballPositionX
0x0001: ballPositionY
0x0002: ballVelocityX
0x0003: ballVelocityY
```

Next I need to write the program instructions starting at address 0 in ROM:

```
ROM   : code       notes
                   updateBallPositionY: ballPositionY = ballPositionY + ballVelocityY
0x0000: 0x0003     load address of ballVelocityY into address register
```

```
0x0001: 0xFC10     load ballVelocityY from memory into data register
0x0002: 0x0001     load address of ballPositionY into address register
0x0003: 0xF088     load ballPositionY + ballVelocityY into ballPositionY in RAM

                   checkBallOffBottom: (ballPositionY >= 480)
0x0004: 0x01E0     load 480 into address register
0x0005: 0xEC10     load value of address register into data register
0x0006: 0x0001     load address of ballPositionY into address register
0x0007: 0xF1D0     load ballPositionY - 480 into data register
0x0008: 0x0010     load ROM address of invertBallVelocityY code into address register
0x0009: 0xE303     jump to invertBallVelocityY code if data register >= 0

                   checkBallOffTop: (ballPositionY < 0)
0x000A: 0x0001     load address of ballPositionY into address register
0x000B: 0xFC10     load ballPositionY into data register
0x000C: 0x0010     load ROM address of invertBallVelocityY code into address register
0x000D: 0xE304     jump to invertBallVelocityY code if data register < 0
0x000E: 0x0012     load ROM address of endOfLoop code into address register
0x000F: 0xEFC7     jump to endOfLoop code

                   invertBallVelocityY: ballVelocityY = -ballVelocityY
0x0010: 0x0003     load address of ballVelocityY into address register
0x0011: 0xFCC8     load -(ballVelocityY) into ballVelocityY in RAM

                   endOfLoop
0x0012: 0x0000     load ROM address of updateBallPositionY code into address register
0x0013: 0xEFC7     jump to updateBallPositionY
```

Exercise

For this week's exercise I'd like you to take a loop from the rules of your game and work out the machine code instructions you'd need to program this rule into the simple CPU we've defined.

Follow my example above as a guide.

Begin by defining the locations in RAM where you want to store any registers you'll use in evaluating your rules. You can just list them sequentially starting at address 0 as in my example above.

Next figure out which instructions you'll need to apply your rules.

Finally, convert the instructions into hexadecimal code and list them out as instructions for the CPU. You can use the table above to help work this out.

Write out your code as in the example above in a plain text file. Submit your files as usual to source control before Sunday. And don't forget to include your work log with your submission.