CHIPS TO SYSTEMS OPEN ASSESSMENT 2003/2004

SYEDUR RAHMAN

CONTENTS

INTRODUCTION

After picking up the assessment sheet on Wednesday week 6 and going through it a couple of times, I had more or less figured out a way of how to go about solving this problem. It seemed pretty simple to me, and as long as we stuck to the basic principles taught in the CTS lectures, I thought this should be a breeze (I was obviously about to be proven wrong later).

My partner and I, who are fortunately like-minded, decided to stick to the minimum specifications (at least to start off with), with efficiency, compactness, cost-reduction (and actually getting something on the scope) being the primary objectives. We decided that we were not going to do the traditional hardware-software split as was suggested with one of the reasons being that we would both benefit most from this course if each of us contributed in both sections.

Although I came up with basic concept of the hardware solution, we decided to connect up all the chips, buses etc. together, i.e. we would be at the lab log ether at the same time throughout the completion of this assignment. I guess that goes with the "true spirit" of team-work (and the only way you make sure the other guys not just lazying around). The software on the other hand, we decided we both would work together on coming up with the basic code e.g. some kind of high-level pseudo-code format and then I would do the actual translation to the Z80 assembly language, replacing instructions with assembly mnemonics, variables with registers/memory locations etc. My partner on the other hand was responsible for coming up with the final animation that we would display, which was an easy decision because of my lack of artistic skills.

I have to admit connecting up the hardware was probably the most difficult part. There were just too many times we spent trying to figure out why our circuits were not working, using the probe to check every single connection on every single pin (not a lot of fun with the 16-bit address lines), replacing every single chip (some of them did turn out faulty) and then simply staring at our circuits hoping for a miracle to occur.

But by the end of week 9, things seemed to be looking up. We were finally getting an image on the screen and getting an animation out of that proved not to be too difficult. However, we had already spent too much times on construction, and therefore we thought it would be wise not to make any further enhancements to the system and instead concentrate on writing this report the last few days.

TECHNICAL DESCRIPTION OF THE PROBLEM

I was taken aback as soon as I first read the word oscilloscope on the assessment sheet. I must admit I had always been quite uncomfortable with working with that device, even while probing a simple power supply voltage. However, I did not have much of a choice but to "change the terms I was on" with the scope, unless I wanted to fail this module.

Like I said before, we had decided to stick to the minimum specifications to start off with. We first found out how the oscilloscope works in X-Y mode and the effects of the Z-input. From what I understood the basic idea was to put in an analogue voltage into the X and Y inputs, to plot a point on the screen and at the same time the Z-input could be used to control the intensity of the beam, i.e. the brightness of the point. After reading the instructions manual of the oscilloscope and pestering the lab technician for quite a while, I more or less understood how the whole X-Y mode works and how the intensity could be used to come up with something like a grey scale image.

Anyway the basic design would mean we would need the Z80 processor to somehow send analogue signals to the X and Y inputs of the scope to plot each point of our image and then give it some time before plotting the next set of points, representing the next frame.

We had decided to go with the bitmap solution, simply because it seemed the easiest one available to us. So the bitmap image would be stored somewhere on some kind of memory device and the main program would then plot each of these points on to their appropriate locations on the screen. However since X and Y accepted analogue values and the Z80 data lines are digital values, a D-to-A converter was required for each of the scope inputs. Initially we decided to stick to a very low resolution (16x16) since one of our main aims was to finish this project soon (if at all), so that we could concentrate on writing up a good report.

On the other hand, we decided to use an analogue device to control the speed input, which meant that we would need an A-to-D converter to send this information to the Z80, which would then speed up or slow down the animation accordingly.

As for holding instructions and our animation, we thought of using a simple EP-ROM device to store them. Using RAM seemed completely unnecessary since none of our programs or data was changing at all throughout the execution of our animation.

PROPOSED HARDWARE DESIGN & JUSTIFICATION

The basic idea was to use the Z80 processor with an EPROM chip (which would hold our program and animation sequence) using memory mapped I/O. Since, we would have three I/O devices at most connected to the Z80, I decided to use 2-4 decoder on the most significant bits of the CPU's address line to select one device at a time and deselect the rest.

Alternatives: We had considered using a discrete I/O scheme as well, but apparently the only advantage it has over memory mapped I/O is that address space is not wasted. So, with a 16-bit address bus using memory mapped I/O on four devices, address lines A15 and A14 are always used to decode what device is being used. Thus, address space is being wasted. However this was not a problem with the hardware that was available to us, i.e. the EPROMs have only 13 address lines, which would imply A15 and A14 were never used anyway. Therefore, we decided to go with the simple memory-mapped scheme as the devices can be placed more or less anywhere in memory and the devices can be accessed just like memory as well. This would also mean that we could use the direct load instructions on the Z80 processor (loading from one memory location to another) to load points directly from the ROM to the DACs (although later we decided against using those instructions).

I know some computer science students who just think of ROM and RAM whenever someone brings up memory in a computer system. Here, like in most embedded systems, there is really no need for any RAM. This is simply because our code really does not need to be changed at all, since we are displaying just one (or more) specific animation(s) and we would program the code such that whatever variable data is needed to stored can be held just using the registers in the Z80.

As I said before, we would start off with trying to display a simple 16x16 image on the screen. This would mean that each point would be 4 bits in X co-ordinates and 4 bits in Y co-ordinates. So, the XY coordinates of each point could be loaded directly on to the data bus from the Z80 at any time. However to convert these digital data into analogue signals (for the oscilloscope inputs), we decided to use two digital to analogue converters, with the higher order bits (B7-B4) going into the X-DAC and the lower orders bits (B3-B0) going into the Y-DAC. So, both the DACs would need to be enabled at the same time, when plotting points. This is done by the 2-4 decoder, as shown in our basic layout diagram.

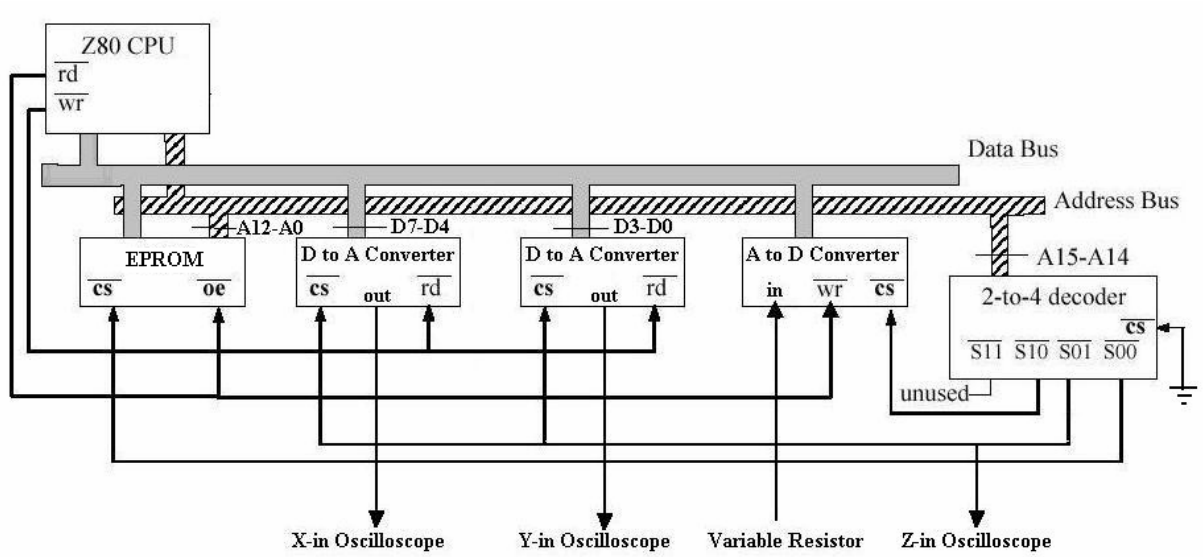
We decided to use a variable resistor with a rotating knob to control the speed of the animation, simply because it seemed the best solution compared with other alternatives e.g. the switches which are simply not user-friendly, and the light-dependent resistor which is simply ridiculous. The output of the variable resistor would obviously need to go through an analogue-to-digital decoder (which would be enabled by the decoder when the speed setting is needed) and into the Z80's data bus.

Initially we had decided to leave the z-input out of the whole solution, since it would require us to write extra instruction or reduce our resolution if we wanted to go by the same sending each co-ordinate in one byte. However, later we were getting trails left behind the beam, since the beam was not being turned off when the DACs were disabled. A simple way of turning down the beam, we thought would be just to turn the beam off when the DACs were not receiving data from the CPU. So, we just decided connect the z-input to the active-low chip enable line of the DACs. This would work out fine since if the beam intensity was turned

down as z would receive a higher voltage, so when the DACs were to be disabled, the z-input would get 5V and the beam would be almost turned off.

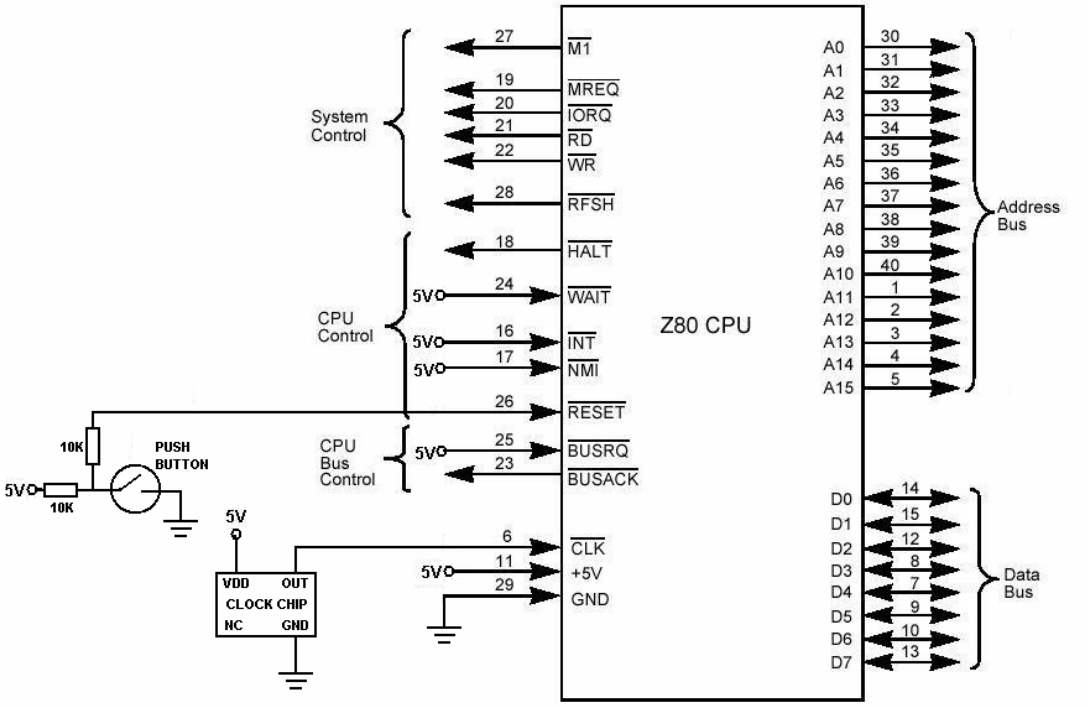
Basic Layout of Hardware

The Overall System



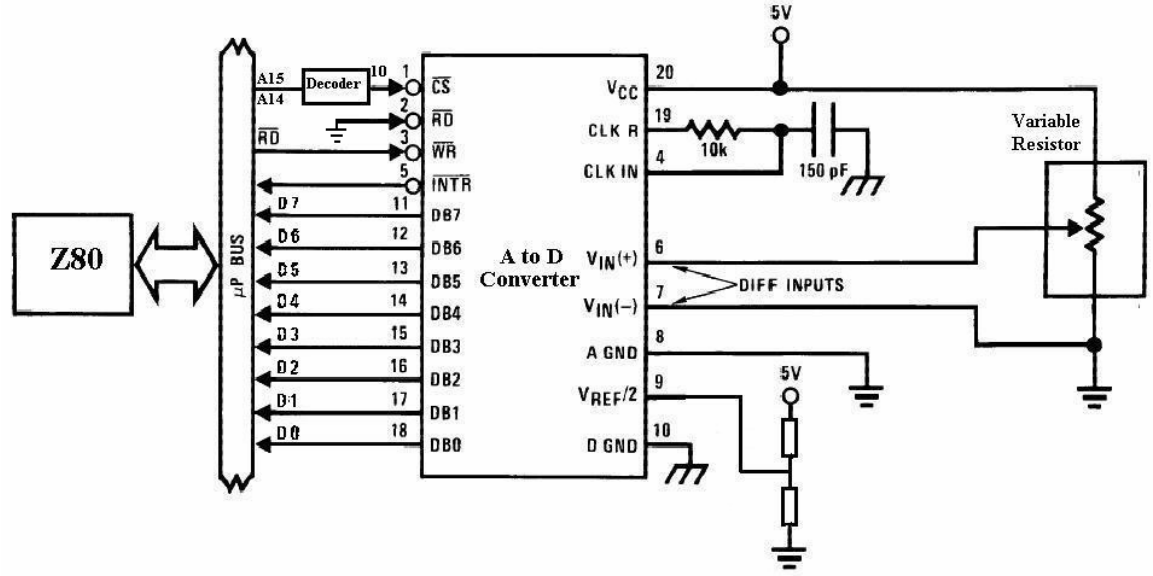
(this diagram is a modified version of the layout of memory mapped systems found on Chris Bailey's CTS website and is also present in my partner's report)

Z80 Basic Connections



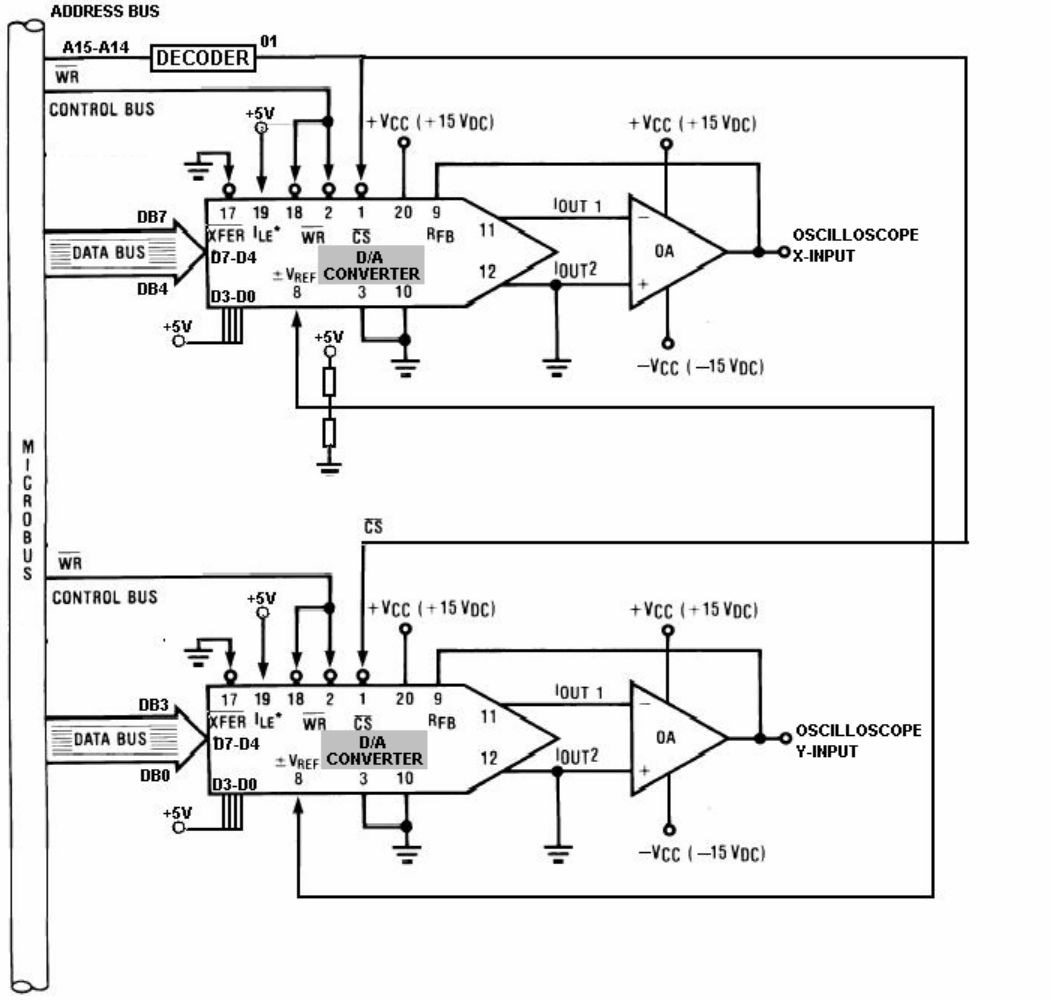
(this diagram is a modified version of that found in the Z80 user's manual)

The ADC connected to the processor and Variable Resistor



(this diagram is modified version of that in the ADC0804LCN's helpsheet on www.national.com)

The DACs connected to the processor



(this diagram is a modified version of that in the DAC0832LCN's helpsheet on www.national.com)

Chips to be used

Model	Name	Qty	Purpose
Z0840004PSC	Z80 CPU	1	The Z80 processor which is essential
SG-431 300228	Crystal Oscillator	1	To provide the clock for the Z80
DAC0832LCN	D/A Converter	2	1 – Perform D/A conversion for X axis 2 – Perform D/A conversion for Y axis
LF356N	Op Amp	2	To amplify analogue signals for each axis on each D/A converter
ADC0804LCN	A/D Converter	1	To convert analogue voltage across the variable resistor to be read in as speed
74LS139N	Dual 2-4 Decoder	1	To select which device the Z80 will be reading/writing data from/to
M27C64A	64 Kbit UV EPROM	1	To store the instructions and the animation

Other components to be used
3 x Breadboards
1 x Push-switch
9 x 22nF capacitors
5 x 1kOhm resistors
1 x Variable Resistor
3 x Oscilloscope probes
Lots of wires

Lab equipment used
1 x Lab DC Power Supply GW-Instek – GPC-M Series
1 x Dual Trace Oscilloscope GW – GOS 658G (50 MHz)

Please note to give smoother signals all chips had a 150nF capacitor across their power and ground lines (which I have not shown in the diagrams).

For further details on how each of the chips are connected (i.e. what pin leads to what device) please see Appendix A.

CONSTRUCTION AND TESTING

We had taken more of a bottom-up approach with construction. The idea was to start with each simple component at a time, connecting them up individually and testing them using simple devices. I have given diagrams of how the individual components were connected up in the final system (see Proposed Hardware and Justification and Appendix A). However I have also written about how these chips were tested individually.

The Digital to Analogue Converters

The first part of construction was to connect each of the DACs with 8-switches controlling the data lines. We had done this before in the CTS lab practicals and we copied the exact same pin connections with 2 new DACs on a new breadboard. Then we connected the op amps (please see diagram), and monitored the output of the op amps on the channels of the oscilloscope. We would expect the point to move down as we increased the binary value going into the Y-DAC and to move to the left as we increased the binary value going into the X-DAC. This was done by simply turning each sets of switches on or off in combinations that represented binary sequences. For the sake of testing we had the chip select, read and write lines always enabled (grounded) and configured the other pins of the DAC such that we were getting a straight through output.

Unfortunately this did not work at first and after spending a few hours painstakingly rewiring every pin, we just decided to start fresh (which was again very painstaking) on a new breadboard and then it did work. We later found out it was because of our first breadboard being faulty, so that was four hours of my life gone that I would prefer not to look back on.

We then tested how the binary values put on the switches affected the X and Y-coordinates on the oscilloscope and we were quite satisfied.

The Analogue to Digital Converter

We connected up the ADC on a separate breadboard, using the usual pin selections shown on the data sheets, so that the device works independently. The variable resistor was connected to the V-in pin of the ADC and the 8 digital outputs went into 8 LEDs, for the initial tests. We would expect the binary number displayed on the LEDs increase or decrease as we turned the knob on the variable resistor up or down.

This didn't seem to be happening initially. As we turned the power supply on we were getting a binary number on the LEDs (e.g. 00000000 if the knob was completely down or 11111111 if it was all the way up or 1000000 if was somewhere in the middle), but when we turned the knob while the power was on, there would be no change in the LEDs. We checked through all the pins again this time (this took up about an hour) and then a miracle (one of the first of many) did occur. I had my finger on the write line of the ADC and my partner was turning the knob up, and it did actually affect the LEDs properly, i.e. they were responding to the change. Initially we did not have a clue as to why that was happening and even contemplated on having a wire going under the table during the demonstration, which I would be touching. But later when I put my finger on the probe of the oscilloscope and discovered it was generating

some kind of sine wave, I figured the write line merely needed some kind of clocking edge to write a value.

So anyway, we got the ADC working properly after that and that was another component successfully constructed.

The Decoder

As I said earlier we would use a 2-4 decoder to select each of the devices (other than the CPU of course) on our system. We simply put one in on the Z80, and put in some inputs (simply using wires connected to 5V and 0V), to check which combination of the 2 inputs, turned which output on and the others off. The device had active low outputs which was very convenient for us since all the other devices' chip select lines are also active low.

The Z80 and the EPROM

We first connected up a clock and a reset switch to the Z80 chip and then connected the data lines to 0s. This would mean after being reset, the Z80 would go through each address, while executing nop instructions. This is exactly what we observed when we probed some of the address and control pins on the processor. We then connected all the data lines to 1s and observed RST instructions being executed.

According the CTS helpsheet, if we were to take out the data lines and stick them to the data pins of a blank EPROM, we should be seeing a similar effect, since all the bits on a blank EPROM are 1s. We did just that and got exactly that result.

Putting it all together

Now it was finally time to combine our separately constructed components into our final system. We started with first connecting the chip select lines of each of the devices to the appropriate lines of the decoder. It was a lot more painstaking though to connect the data lines to all the four devices.

Please note that the DACs were made to share the same chip select line, since they are activated at the same time to send X and Y coordinates to the oscilloscope. Moreover, each DAC just had four data lines going into its higher order bits, i.e. bits 7-4 of the data bus went into D7-D4 of the X-DAC and bits 3-0 of the data bus went into D7-D4 of the Y-DAC. The lower order bits of both the DACs were pulled to 1s. For some strange reason my partner suggested using the lower order bits of the DACs to get input from the CPU but after a while I managed to explain to him that using the lower order bits would decrease the sensitivity of our DACs to the changed in digital data sent from our processor, or something like that.

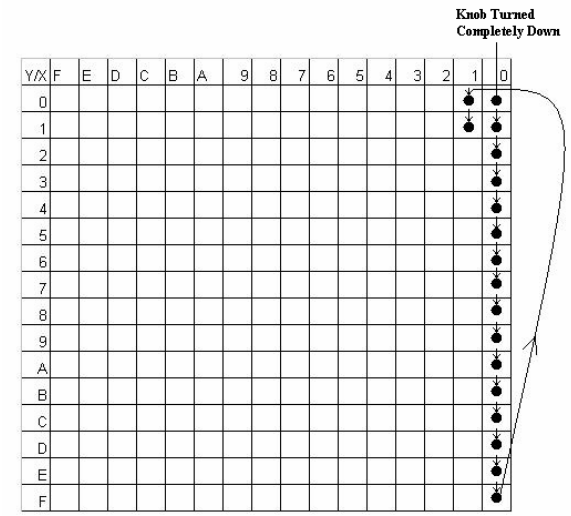
The ADC's chip select line went into the third output of the decoder (i.e. the one that gets activated when A15-14 = 10) and we tied its read lines and writes lines appropriately to the CPU.

The first program we tested was simply something that would plot all points on the screen. This simply incremented the A register and started loading the values on to the DACs, so each

and every possible coordinate would get plotted every time A went from 00 to FF. However we did not seem to be getting that at all, in fact we were getting nothing on the screen.

After trying (and failing) to figure out what was wrong for a few more hours, we connected our Z80 up with the Z80 probe/analyser to check what instructions were being executed. We discovered it was not at all anything we had compiled. We later checked the address and data line connections between the EPROM and the Z80 and discovered that we had switched around 2 data lines, which would mean the first instruction it was fetching was not right and therefore the CPU was jumping to some random state by executing that instruction. After fixing that we were getting all our points on the screen.

Then we tested if the ADC was connected properly by simply getting it the CPU to load the value from the ADC onto one of the registers and then writing that value on to the DACs. As we turned the knob up, we saw the point on the screen moving down (since the Y coordinates were changing) and when it moved to the sixteenth level dot on the screen, it then moved back to the top dot, but only one step to the left. This is exactly what he had expected, since the Y represented the lower orders bit and X the higher order bits, so after co-ordinates 0,F, it should go to coordinates 1,0 (it moves down and to the left because our op-amps are set to invert the output)



This also proved to be a good test of whether our DACs were working properly, since it showed whether the points we wanted to plot were the ones exactly being plotted.

Later we programmed EPROMs to display a few pictures on the screen (using simple load instructions) and eventually animations (please see software design section). Then finally, we wrote a program that uses the value from the ADC to control the speed of our animation.

The Z-input

While building the hardware we were really not sure whether we needed a z-input at all. So, we had ignored it through out the earlier construction phases. Other systems needed the z-input since some of them may be sending information to each DAC separately (while using the z-input to have the oscilloscope beam turned off) and then when the 2 DACs have received the X and Y co-ordinates, the beam would be turned back on using the z-input. However, our system sends X and Y co-ordinates at the same time to the oscilloscope, so the

beam can be left on throughout this period. However we do not want the beam on, when the DACs have been disabled, since during this time the op amps would give signals of –5V, making the beam go back to (0,0) between plotting two points. This did indeed leave some kind of faint trail on most of our images. The simplest solution to alleviate this problem was to make sure that the beam is on only when the DACs are enabled. Since the z-input uses some kind of inverted input (i.e. intensity is highest when the input is 0), all we had to do was connect the z-input to the chip-select of the DACs. So when the DACs got enabled (chip-select going to 0V), the z-input turned the intensity on and when the DACs got disabled (chip-select going to 5V) the z-input turned the intensity down (to almost off).

For further details on how each of the chips are connected (i.e. what pin lead to what device) please see Appendix A.

SPECIFICATION AND DESIGN OF THE SOFTWARE

Plotting an image

We both first came up with the simplest solution to plot points on the oscilloscope's screen. Since we were using memory mapped I/O, with the DACs activated when address lines A15 and A14 went 0 and 1 respectively, every time we used a load instruction on a memory location 01XXXXXXXXXXXXXXX, the DACs would get enabled and would read whatever was on the data bus.

Since the Z80 assembly language uses hex values, we could just use something like "ld (0x4000), 0xXY" (where X and Y are hexadecimal values that represent our X and Y co-ordinates respectively) to plot a point on the screen. This is because memory location 4000h is the first location of the 01XXXXXXXXXXXXXXX, although it would not make a difference if we used any values between 4000-7FFF.

However, since the Z80 does not allow immediate addressing while loading to memory locations we would first have to store XY (which is one byte) on to a register. So, the instruction for plotting (X,Y) on the screen would be like

```
ld a, XY
ld (0x4000), a
```

(using the A register to temporarily store the co-ordinates)

So our image would just be a simple sequence of this pair of instructions, each representing a point, and they would altogether make a frame.

Reading the value off the ADC

Since the ADC is connected to the third output of the decoder it gets enabled when A15 and A14 on the address bus are 1 and 0 respectively, i.e. the address is of the form 10XXXXXXXXXXXXXXX. With our memory mapped I/O system, the simplest way to read off the ADC would be to have an instruction like "ld a, (0x8000)". Similarly for the ADC any address value between 8000h and BFFFh would have made no difference.

Getting an animation

The initial idea to get an animation (ignoring the speed setting for now) was to display each frame by plotting each of its points using pairs of ld instructions and then repeating this process of displaying a frame a certain number of times before going on to the next frame. In pseudo-code it would look like this

```
For l=FFh to 01h

        ld a, f1_xy1
        ld (0x4000), a
              :
```

```
        :
        :
    ld a, f1_xyn
    ld (0x4000), a
```

Next

(where fm_xyn represents the co-ordinates of the n-th point on the m-th frame).

So frame 1 would be repeated FF (256) times and after this frame two would proceed and so on.

For l=FFh to 01h

```
    ld a, f2_xy1
    ld (0x4000), a
        :
        :
        :
    ld a, f2_xyn
    ld (0x4000), a
```

Next

A very simple way of changing the speed would be to change the starting value of l. So if l started at 7F instead of FF, the frame would be repeated half the number of times and so it would move on to the next frame twice as fast, thus speeding up the animation two times.

So in order to control the speed all we would have to do is make it so that the (FF-speed) is somehow put in as the starting value of l. This would mean the higher the speed setting, the fewer times the frame is repeated and the faster the animation. It is not difficult at all to do a speed=FF-speed, since this just means inverting the bits on the register holding the speed.

So as a result our speed module looks like this

```
Ld (8000h), h
cpl h
```

So now h would hold the number of times each frame should be repeated.

We did not have to set up any initial conditions or register values, since the reset CPU does just that and the first module we run is speed before display, so our speed setting does not have to be initialised to anything either.

The Display Module

One alternative that was available to us was to store the whole image as bits (since we are doing a “2-colour” image) in some part of the ROM and then write a program that would take each bit and while keeping track of what co-ordinate that represents and then plotting that

point on the oscilloscope if that bit happens to be a 0. However the Z80 can fetch one whole byte at a time from the EPROM and not just one bit. So, we would have to fetch eight points at the same time, and then test each bit individually. At the same time, we would have to keep a register which would hold our current XY co-ordinates (that gets incremented every time the next bit is being read) and if the corresponding bit being tested is a 1, XY is sent to the oscilloscope.

So, lets say if we started at location 1000h, the bits that would represent the status for each pixel of the first frame would be as follows

Address	1000								1001							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Co-ordinate	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0.7	0,8	0,9	0,10	0,11	0,12	0,13	0,14	0,15

:
:

Address	100F								1010							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Co-ordinate	7,8	7,9	7,10	7,11	7,12	7,13	7,14	7,15	8,0	8,1	8,2	8,3	8,4	8,5	8,6	8,7

:
:

Address	1001F								1020							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Co-ordinate	F,0	F,1	F,2	F,3	F,4	F,5	F,6	0.7	F,8	F,9	F,10	F,11	F,12	F,13	F,14	F,15

So, a total of 32 bytes would be needed per frame. The animation we had been considering using takes up 19 frames, so addresses 1000h-125Fh (i.e. 1000h + 19d*32d -1) would hold our animation sequence.

We have a total of 3000 bytes free to hold our animation, so technically this means we could have had (3000h/32d =) 190 frames if each frame is of resolution 16x16.

Basically the first frame would look like this

```
XY = 00h
MemP = 1000h

For I = 1 to 32
      Ld  cbyte, MemP
      ‘ checking each bit before plotting
      if cbyte(7) = 1 then ld (4000h), XY
      XY++
      If cbyte(6) = 1 then ld (4000h), XY
      XY++
      If cbyte(5) = 1 then ld (4000h), XY
      XY++
      If cbyte(4) = 1 then ld (4000h), XY
      XY++
      If cbyte(3) = 1 then ld (4000h), XY
      XY++
      If cbyte(2) = 1 then ld (4000h), XY
      XY++
      If cbyte(1) = 1 then ld (4000h), XY
      XY++
      If cbyte(0) = 1 then ld (4000h), XY
```

```
      MemP++            ' preparing to read next byte
Next
```

So as you can see the first byte fetched from memory location 1000 (the start of our animation) holds the status for co-ordinates 0,0 to 0,8 (and 1001 hold co-ordinates for 0,9 to 0,15 and so on). Each of these bits are tested, just when XY holds that the value of the co-ordinates and if the bit is a 1, the point is plotted on the screen. Since each image has (16x16 =) 256 points, a total of (256/8 =) 32 bytes will be taken up.

A simpler alternative to doing this (that I initially considered) was to hold the co-ordinates of only the points to be plotted on the EPROM. So, each frame would merely be a sequence of load direct instructions loading from the EPROM to memory location 4000.

However this would mean each point would now take up one byte instead of one bit. So, even if only half the points were to be plotted per frame on average, this would take up four times as much space on the EPROM.

Our first program on the other hand, which just load instructions on immediate values embedded within the code would technically take up six times the space, since each extra point would now mean two extra load instructions. So it was obvious why we had to abandon the idea.

So the main display program would then display this frame (FF-speed) number of times, and each time it displays the frame once it would need to reset the memory pointer to the start of the frame, which can be done simply by saying MemP=MemP – 32 (since each frame is 32 bytes).

It would do this for each of the frames and then finally stop and return to the main procedure.

So when the frame has been repeated the required number of times, the MemP which has been reset to the start of the current frame, is simply set to the next frame by doing MemP = MemP + 32.

```
MemP = 0x1000

For K = 1 to 19

      For J= 1 to (FF-speed)

            XY = 00h
            MemP = 1000h

            For I = 1 to 32
                  Ld  cbyte, MemP
                  ' checking each bit before plotting
                  if cbyte(7) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(6) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(5) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(4) = 1 then ld (4000h), XY
```

```
                  XY++
                  If cbyte(3) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(2) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(1) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(0) = 1 then ld (4000h), XY
                  MemP++

            Next

            MemP = MemP - 32        ' go back to start of current frame

      Next

      MemP = MemP + 32              ' go to start of next frame

Next
```

However this turned out to be a problem since a total of 9 registers were needed to perform these operations.

I found a better way of executing the same set of instructions using less registers. First of all, there is no need for I to count from 1 to 32, since when I = 32, XY goes to FFh and then gets incremented to 00h. So, at the end of every I-loop we could just check whether XY is 0, thus negating the need for an extra register to keep track of I.

Instead of repeating the frame part using a for-next loop using a variable K, we could just check at the end of every frame repeated whether it was the last frame. We know that our animation has 19 frames and each frame is 32 bytes. Since the animation starts at address 1000h, the last byte of the animation would be 1000h + 19d*32d – 1 = 125F. That is, when MemP has been incremented to 1260, we know the animation has ended and we can return to the main procedure.

```
MemP = 0x1000
XY = 00h

Repeat

      For J = 1 to (FF-speed)


            MemP = 1000h

            Repeat
                  Ld  cbyte, MemP
                  ' checking each bit before plotting
                  if cbyte(7) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(6) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(5) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(4) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(3) = 1 then ld (4000h), XY
                  XY++
```

```
                If cbyte(2) = 1 then ld (4000h), XY
                XY++
                If cbyte(1) = 1 then ld (4000h), XY
                XY++
                If cbyte(0) = 1 then ld (4000h), XY
                MemP++
          Until XY = 00

          MemP = MemP - 32                    ‘  go  back  to  start  of
    current frame

    Next

    MemP = MemP + 32                          ‘ go to start of next frame

Until MemP = 1260h
```

Registers used

Register	Variable	Purpose
BC	MemP	Address of byte being read
H	FF-speed	Repeats required per frame
L	J	Repeats left for current frame
A	XY	Co-ordinates of current bit being tested
D	Cbyte	Current byte read (Contents of MemP)
E		register used to temporarily hold the value of A since some load operations can only be performed on register A

(please note that the pseudo-code is shared by my partner and myself and is present in the exact same form in both our reports)

In the final code the load instructions that write to the DACs were interleaved with 8 nop instructions in between to give the DACs enough time between two points so that they could be plotted properly.

Since we do not need any initialisation procedures, our main module (which starts at location 0000h) would simply look like this:

Call Speed
Call Display
JP 0000h

However, since we were only using ROM and no RAM, we could not call procedures, since we had no space for a stack (to store return addresses). Since our main program is simply a sequence of instruction any way, with no nested procedure calls, we thought it would not make a difference if we simply pasted the body of the speed procedure followed by the body of the display and then putting a final instruction saying jump to beginning. It seemed rather pointless having a RAM device just to store the stack, when it is quite unnecessary to start with and we did however test the modules separately first.

Please see Appendix C for the final annotated assembly code

SOFTWARE TESTING

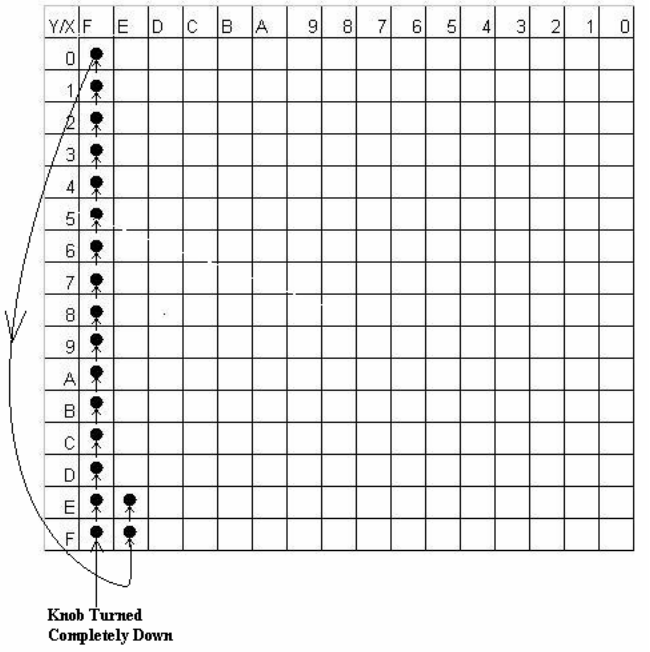
Each module before being written on to the EPROM was tested separately before the system was tested as a whole. All initial software tests were conducted on the Z80 emulator found on the linux machines at the university's labs. The emulator allows the user to load an object program and then step through the instructions while monitoring the contents of the memory and registers at the same time.

Module Testing

Speed

The speed module simply reads information from memory location 8000, puts it on to the l register and then inverts the bits. Testing it on the emulator just involved putting a value on to location 8000, then running the program and checking the registers (using the display command) to see if the l register had been updated with contents of 8000.

The module was also tested on hardware, with a main procedure that keeps calling speed and then writes the contents of register l onto memory location 4000 (i.e. the DACs). As we turned the knob on the variable resistor up, we saw the point on the screen moving up (since the Y coordinates were changing) and when it moved to the sixteenth level dot on the screen, it then moved back to the bottom dot, but only one step to the right. This is exactly what he had expected, since the Y represented the lower orders bit and X the higher order bits, co-ordinates FF represented were the bottom left of the screen but since we inverted the bits (because this is actually repeats-per-frame and not speed), this was the point first plotted when the variable resistor was completely turned down (i.e. it was reading 00).



Display

The emulator was once again used on the display procedure. First it was tried with the simple version of our display procedure, which for each point on a frame had pairs of load instructions, which directly loaded co-ordinates onto memory location 4000. I set the default

speed setting to a feasible number (between 5 and 10). At the beginning of every frame this value would be stored in one of the registers (h). As each frame was repeated I observed the value of h decrementing and then when h came down to zero, the instructions for the next frame were executed.

It was now time to test the final version of the display procedure. For this I first compiled the object code and then loaded it into memory location 0000. I then wrote a binary file with the status of each of the pixels for each frame and then loaded that file at memory location 1000 (the start pointer for the display procedure to read points from). I stepped through the instruction and observed that address 4000 was only getting written to when there was a 1 on the bit being tested and then checked with the pictures of each frame I had on paper, whether the co-ordinates being plotted were right.

It was more painstaking to check whether each frame was being repeated l number of times, because realistically l would be around 256, however I used values between 2 and 5 to test it. I also needed to check whether program was moving on to the next frame and indeed it was, since the 4000 address was being updated with the co-ordinates of the new frame.

Then we finally decided to test the hardware using the display procedure with a considerably low speed setting (i.e. a high value of l). The animation of the screen came out pretty well and we were more than satisfied.

System Testing

The program was this time compiled and then tested on the emulator once again to check whether each of the modules was getting accesses properly. It was indeed first running the speed procedure reading a value of address 8000 (we had put in large values since that would mean the frames would be repeated fewer times). As it goes with all testing procedures, we tried out the extreme values for the ADC input such as 00000000 (which is more or less a standstill image, possibly taking minutes to change to the next frame, and obviously we gave up on stepping through the instructions for that after a about 50 instructions) and 11111111. We expected the latter to give an image so fast that we would not be able to see on the screen. However this was not the case, since 11111111 got inverted to 00000000 as the initial repeats left. But the program doesn't check whether the repeats left has gone down to zero before going through the loop once (which decrements it). As a result the repeats left goes down from 00000000 to 11111111 and this means the image actually becomes VERY slow instead of VERY fast.

Then we finally burnt the EPROM with the program starting at address 0000 and the animation starting at address 1000. We put it into the Z80 and we saw a proper animation on the screen. Moreover the variable resistor actually did turn the speed up or down proportionately.

SUMMARY OF WORK AND CONCLUSION

It seems that we had more or less accomplished what we set about to do initially. We did go beyond the minimum specifications and ended up producing an animation of 19 frames each of 16x16 resolution. The design phase of this assignment did not take much at all, most of it we had figured out in less than a day and made some very minor modifications as we went along. However, it was the construction that simply killed us. There were simply too many things not going our way, burnt chips, faulty breadboards and so on. It also just took too long to find out if we had some connection the other way around or if one wire was touching the other. In the end, we were just glad that everything worked out and we got an animation at all on the screen. So, we had to scrap a lot of enhancements we had in mind, which I will be discussing now.

First of all, I would have liked to have a 256x256 image using 8 bits per axis. That would have just required me to write some simple instructions that would get one of the DACs to latch one of the axis (since only 8 bits can be sent at one time) while the beam was turned off, then send the other axis over to the other DAC and turn the intensity up to plot a point of the image. Obviously, we took so much time to getting the DACs working at all, we had to scrap this idea.

Another further enhancement I would have liked to implement is to have a separate EPROM chip to hold the images of the animation, instead of holding it on the same EPROM. This would not be a problem at all since we do have an extra line from the decoder available (the program could start reading pixels from location C000 instead of 1000) and this would also mean more frames could be stored on the second EPROM since it is not holding the program. This would also make our system similar to a video cassette player, with the second EPROM being like the video cassette holding the animation to be played. This would not have been difficult at all to implement, but this assignment requires us to be able to display just one animation and not build some kind of console system, thus wasting another EPROM when the first one was more than enough to hold both our program and animation, seemed quite inefficient.

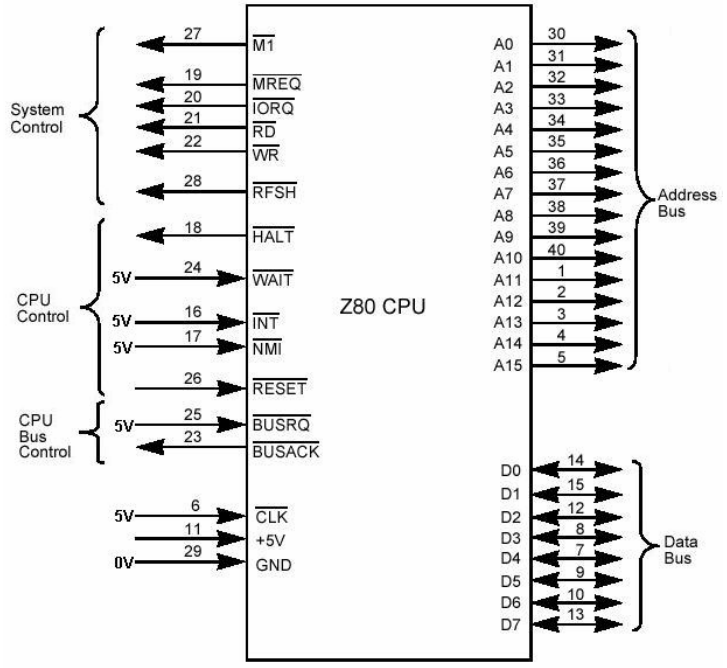
In conclusion, I would like to say that despite all those moments I have simply called painstaking (for lack of a better word), this was quite an enjoyable experience for both me and my partner, especially when we had our animation running by the last lab practical session and watched some other pairs shouting at their circuits and cursing the oscilloscope. We realise we could have come up with a much more sophisticated system, but our initial idea was to just somehow build the minimum specification system (although we did more than that) and making it as efficient as possible (which I think we did accomplish as well).

APPENDICES

A – Chips Used and Pin Connections

(chip diagrams taken from National Semiconductor's website)

Z80 CPU - Z0840004PSC

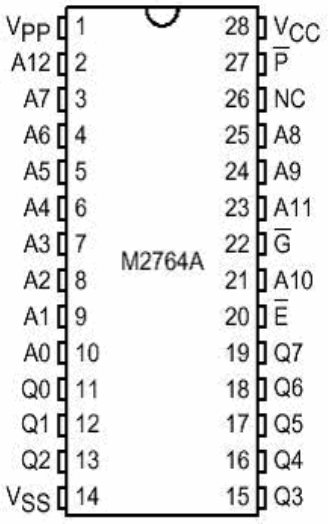


Pin	Name	Connected to Device-pin (pin no)
27	-M1	
19	-MREQ	Decoder-1G (1)
20	-IORQ	
21	-RD	EPROM-G (22)
22	-WR	
28	-RFSH	
18	-HALT	
24	-WAIT	5V
16	-INT	5V
17	-NMI	5V
26	-RESET	Push Switch (active low)
25	-BUSRQ	5V
23	-BUSACK	
6	-CLK	Clock-out (3)
11	+5V	5V
29	GND	0V
30	A0	EPROM-A0 (10)
31	A1	EPROM-A1 (9)
32	A2	EPROM-A2 (8)
33	A3	EPROM-A3 (7)
34	A4	EPROM-A4 (6)

35	A5	EPROM-A5 (5)
36	A6	EPROM-A6 (4)
37	A7	EPROM-A7 (3)
38	A8	EPROM-A8 (25)
39	A9	EPROM-A9 (24)
40	A10	EPROM-A10 (21)
1	A11	EPROM-A11 (23)
2	A12	EPROM-A12 (2)
3	A13	
4	A14	Decoder-1A (3)
5	A15	Decoder-1B (2)
14	D0	EPROM-Q0 (11), DACY-DI4 (16), ADC-DB0 (18)
15	D1	EPROM-Q1 (12), DACY-DI5 (17), ADC-DB1 (17)
12	D2	EPROM-Q2 (13), DACY-DI6 (18), ADC-DB2 (16)
8	D3	EPROM-Q3 (15), DACY-DI7 (19), ADC-DB3 (15)
7	D4	EPROM-Q4 (16), DACX-DI4 (16), ADC-DB4 (14)
9	D5	EPROM-Q5 (17), DACX-DI5 (17), ADC-DB5 (13)
10	D6	EPROM-Q6 (18), DACX-DI6 (18), ADC-DB6 (12)
13	D7	EPROM-Q7 (19), DACX-DI7 (19), ADC-DB7 (11)

64 Kbit UV EPROM - M27C64A

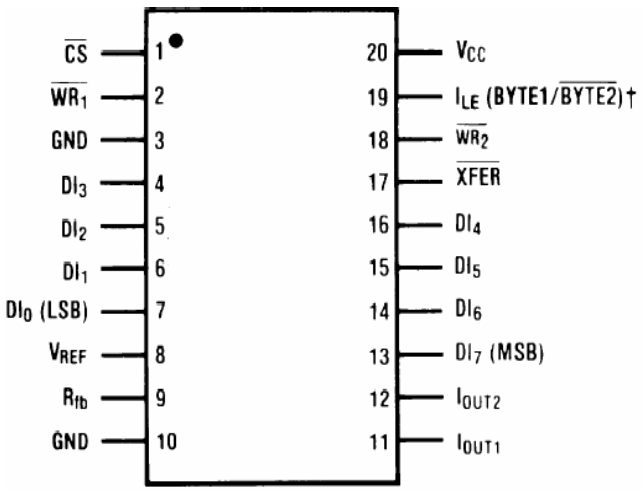
Pin	Name	Connected to Device-pin(pin no)
1	Vpp	0V
2	A12	Z80-A12 (2)
3	A7	Z80-A7 (37)
4	A6	Z80-A6 (36)
5	A5	Z80-A5 (35)
6	A4	Z80-A4 (34)
7	A3	Z80-A3 (33)
8	A2	Z80-A2 (32)
9	A1	Z80-A1 (31)
10	A0	Z80-A0 (30)
11	Q0	Z80-D0 (14)
12	Q1	Z80-D1 (15)
13	Q2	Z80-D2 (12)
14	Vss	0V
15	Q3	Z80-D3 (8)
16	Q4	Z80-D4 (7)
17	Q5	Z80-D5 (9)
18	Q6	Z80-D6 (10)
19	Q7	Z80-D7 (13)
20	-E	Decoder-out1 (4)
21	A10	Z80-A10 (40)
22	-G	Z80-RD (22)
23	A11	Z80-A11 (1)
24	A9	Z80-A9 (39)



25	A8	Z80-A8 (38)
26	NC	
27	-P	5V
28	Vcc	5V

D/A Converter for X - DAC0832LCN

Pin	Name	Connected to Device-pin(pin no)
1	-CS	Decoder-out2 (5)
2	-WR1	Decoder-out2 (5)
3	GND	0V
4	DI3	5V
5	DI2	5V
6	DI1	5V
7	DI0	5V
8	Vref	2.5 V (made by using a potential divider from 5V using two 1kohm resistors)
9	Rfb	OpAmpX (6)
10	GND	0V
11	Iout1	0V, OpAmpX (3)
12	Iout2	OpAmpX (2)
13	DI7	Z80-D7 (13)
14	DI6	Z80-D6 (10)
15	DI5	Z80-D5 (9)
16	DI4	Z80-D4 (7)
17	-XFER	0V
18	-WR2	0V
19	Ile	1
20	Vcc	5V



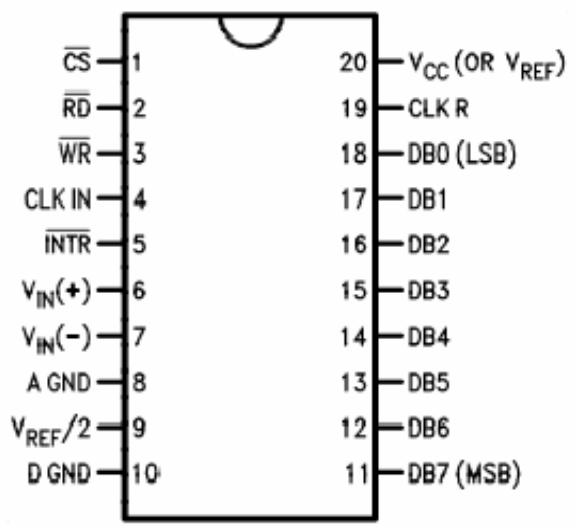
D/A Converter for Y - DAC0832LCN

Pin	Name	Connected to Device-pin(pin no)
1	-CS	Decoder-out2 (5)
2	-WR1	Decoder-out2 (5)
3	GND	0V
4	DI3	5V
5	DI2	5V
6	DI1	5V
7	DI0	5V
8	Vref	2.5 V (made by using a potential divider from 5V using two 1kohm resistors)
9	Rfb	OpAmpY (6)
10	GND	0V
11	Iout1	0V, OpAmpY (3)
12	Iout2	OpAmpY (2)
13	DI7	Z80-D3 (8)

14	DI6	Z80-D2 (12)
15	DI5	Z80-D1 (15)
16	DI4	Z80-D0 (14)
17	-XFER	0V
18	-WR2	0V
19	Ile	1
20	Vcc	5V

D/A Converter - ADC0804LCN

Pin	Name	Connected to Device-pin (pin no)
1	-CS	Decoder-out3 (6)
2	-RD	0V
3	-WR	Z80-RD (21)
4	CLKIN	150pf to 0V
5	-INTR	ADC-WR (3)
6	Vin+	Variable Resistor
7	Vin-	0V
8	A GND	0V
9	Vref/2	2.5 V (made by using a potential divider from 5V using two 1kohm resistors)
10	D GND	0V
11	DB7	Z80-D7 (13)
12	DB6	Z80-D6 (10)
13	DB5	Z80-D5 (9)
14	DB4	Z80-D4 (7)
15	DB3	Z80-D3 (8)
16	DB2	Z80-D2 (12)
17	DB1	Z80-D1 (15)
18	DB0	Z80-D0 (14)
19	CLK R	10kohm to ADC-CLKIN (4)
20	Vcc	5V

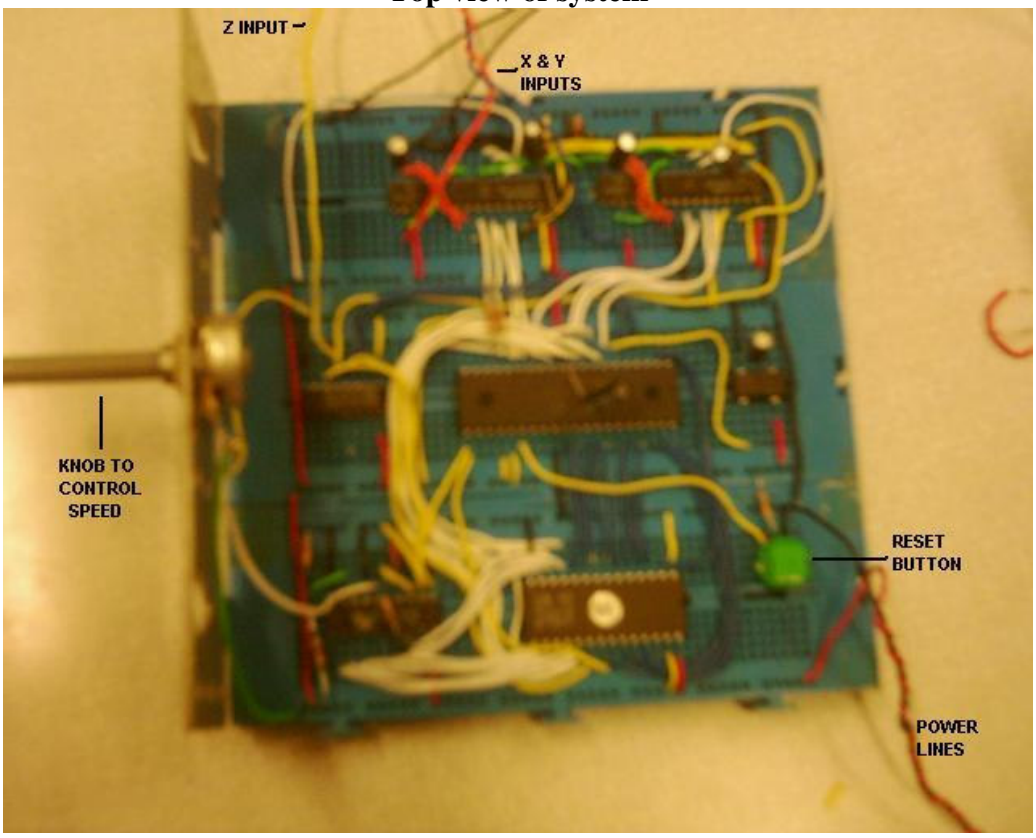


2-4 Decoder

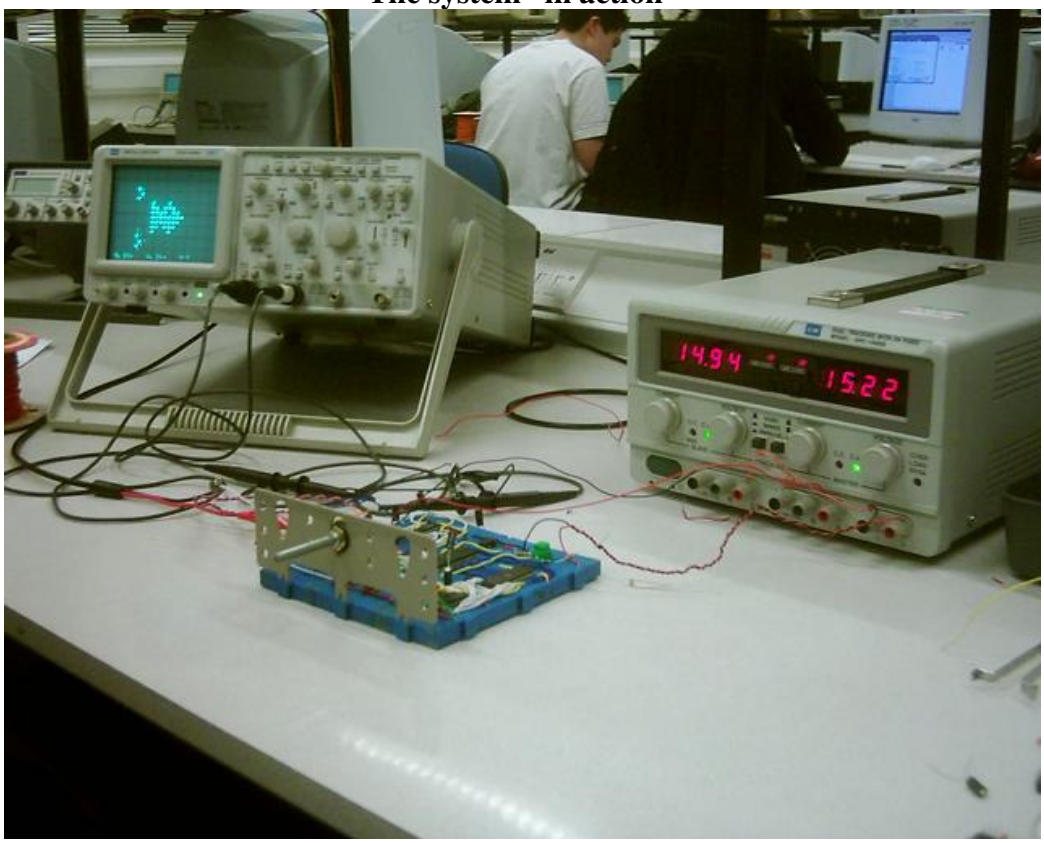
Pin	Name	Connected to Device-pin (pin no)
1	1G	Z80-MREQ (19)
2	A1	Z80-A1 (5)
3	B1	Z80-A14 (4)
4	X1	EPROM-E (20)
5	X2	DACX-CS (1), DACY-CS (2)
6	X3	ADC-CS (1)
7	X4	
8	GND	

B – Pictures of the System

Top view of system



The system "in action"



C – The Code

Instructions	Annotations
Speed:	
ld a, (8000)	Loads value from ADC onto A
cpl a	Inverts bits of A
ld l, a	Loads repeats left on to l
jp z, iszero	
jp display:	
iszero:	If l = 0 then changes it to 1
inc l	
Display:	
ld b, 0x10	Set bc to start from 1000
ld c, 0x00	
ld d, 0x00	Set d = 0
ld h, l	Loads h with no of times frame is to be repeated
point0:	
ld e, a	Loads contents of memory location held by bc on to d
ld a, (bc)	(using e to temporarily hold the previous value of a,
ld d, a	since only can be used to perform indirect loads)
ld a, e	
bit 7, a	Checks MSB, if 1 then sends point to DACs
nop	Nops to give DAC enough time to respond before next
nop	co-ordinate
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point1	
ld (0x4000), a	
point1:	
inc a	Changes to next co-ordinate
bit 6, a	Checks bit, if 1 then sends point to DACs
nop	Nops to give DAC enough time to respond before next
nop	co-ordinate
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point2	(otherwise moves to next point)
ld (0x4000), a	
point2:	

inc a	
bit 5, d	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point3	
ld (0x4000), a	
point3:	
inc a	
bit 4, d	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point4	
ld (0x4000), a	
point4:	
inc a	
bit 3, d	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point5	
ld (0x4000), a	
point5:	
inc a	
bit 2, d	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point6	
ld (0x4000), a	
point6:	
inc a	
bit 1, d	

nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, point7	
ld (0x4000), a	
point7:	
inc a	
bit 0, d	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
nop	
jp z, nextbyte	
ld (0x4000), a	
nextbyte:	When all 8 bits of each byte are written moves on
inc bc	to next byte
inc a	If a becomes 0 that means all points on current frame have been read, so goes on to repeat the frame
jp z, repeatframe	
repeatframe:	
dec h	No of repeats left decremented
jp z, newframe	If no repeats left it plots next frame
ld a, c	Otherwise repeats current frame
sub 0x20	(done by subtracting 32 from next point to be read)
ld c, a	
jp c, nocarry	
Dec b	
nocarry:	
jp startframe	
newframe:	
ld a, b	For a new frame, it first checks whether it is end of animation
Sub 0x12	
jp z, checkend	
ld h, l	If not ended, it reloads repeats left and starts next frame
jp startframe	
checkend:	Checks if animation ended i.e. bc = 1260
ld a, c	
sub 0x60	
jp z, endofdisplay	
ld h, l	If not ended, it reloads repeats left and starts next

	frame
jp startframe	
endofdisplay:	If animation ended, it goes to start of program
jp 0x0000	

The data on the animation (each pixel as a 1 or 0) was stored starting from memory location 1000h on the EP-ROM with each frame taking up 32 bytes (256 pixels) and the entire animation (which is 19 frames) taking up 608 bytes (260h).

Pseudo-Code

```
MemP = 0x1000
XY = 00h

Repeat

      For J = 1 to (FF-speed)


            MemP = 1000h

            Repeat
                  Ld  cbyte, MemP
                  ‘ checking each bit before plotting
                  if cbyte(7) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(6) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(5) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(4) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(3) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(2) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(1) = 1 then ld (4000h), XY
                  XY++
                  If cbyte(0) = 1 then ld (4000h), XY
                  MemP++
            Until XY = 00

            MemP = MemP – 32                    ‘  go  back  to  start  of
      current frame

      Next

      MemP = MemP + 32                          ‘ go to start of next frame

Until MemP = 1260h
```

Registers used

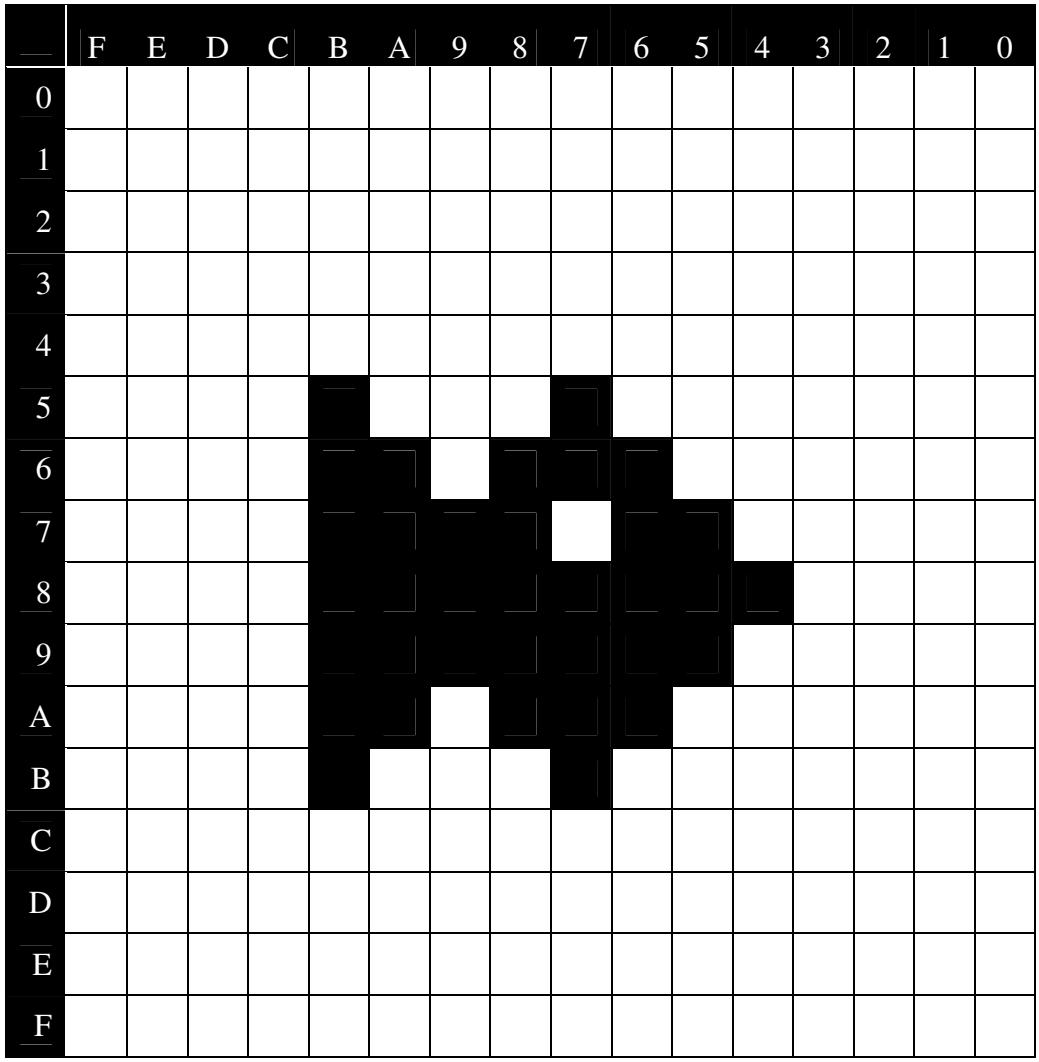
Register	Variable	Purpose
BC	MemP	Address of byte being read
H	FF-speed	Repeats required per frame

L	J	Repeats left for current frame
A	XY	Co-ordinates of current bit being tested
D	d	Current byte read (Contents of MemP)
E		register used to temporarily hold the value of A since some load operations can only be performed on register A

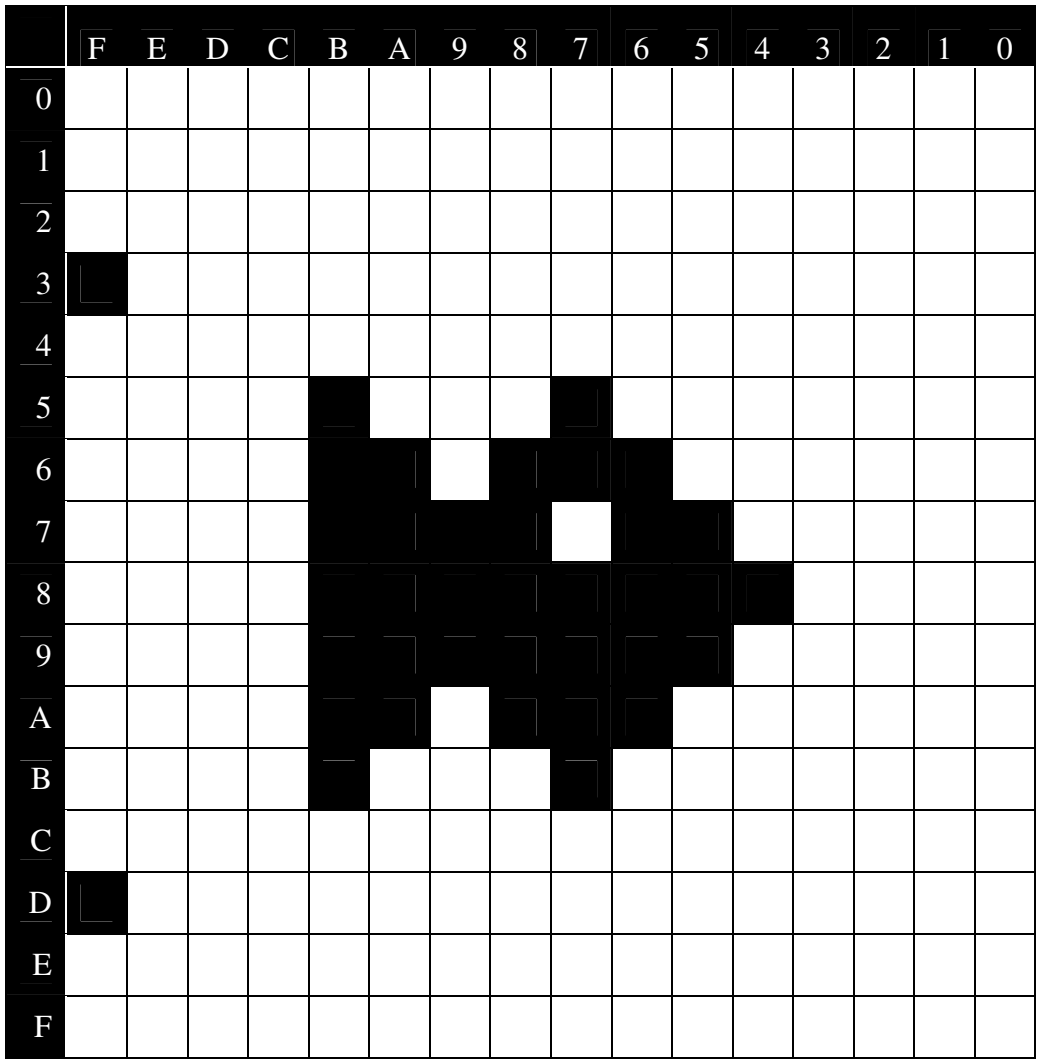
D – The Animation

(The animation was done by my partner and is thus present in both our reports)

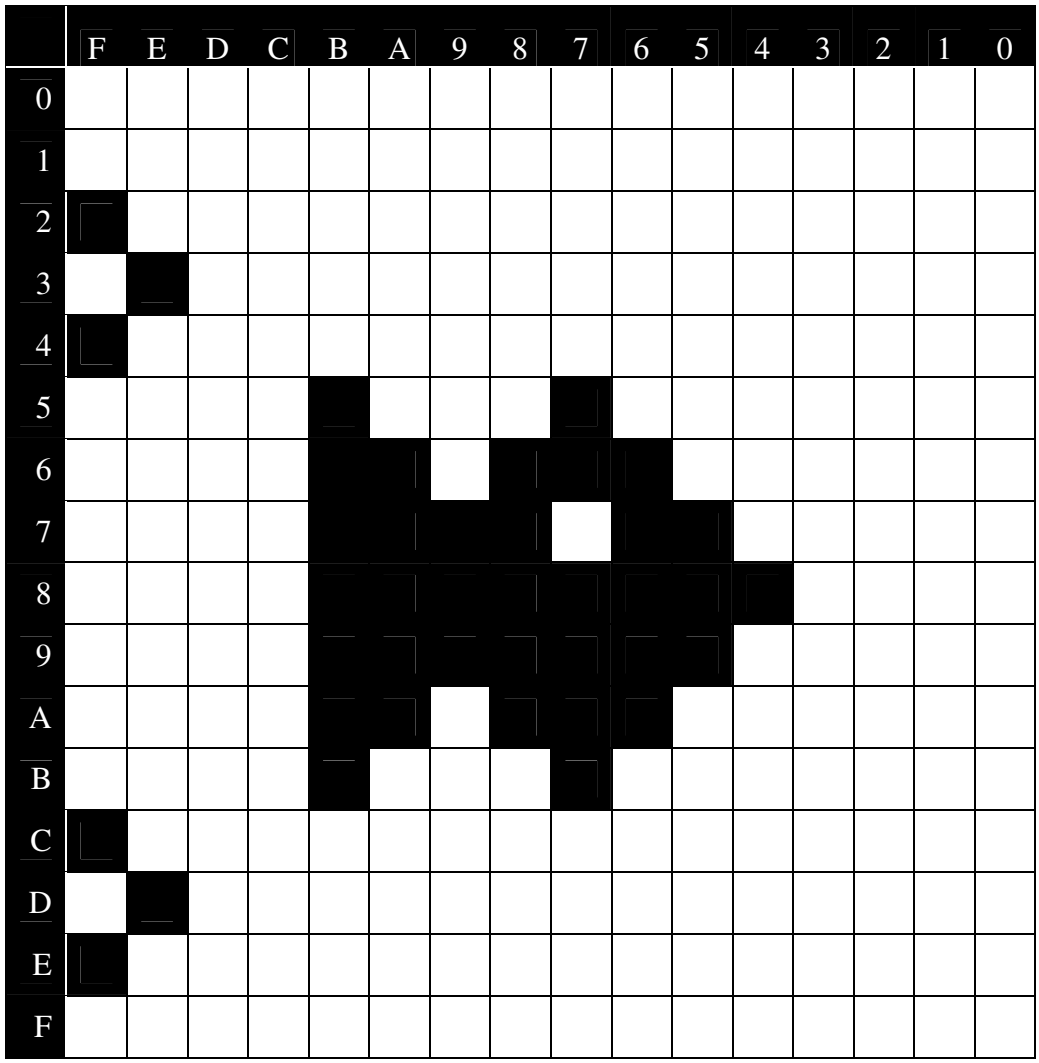
Frame 1



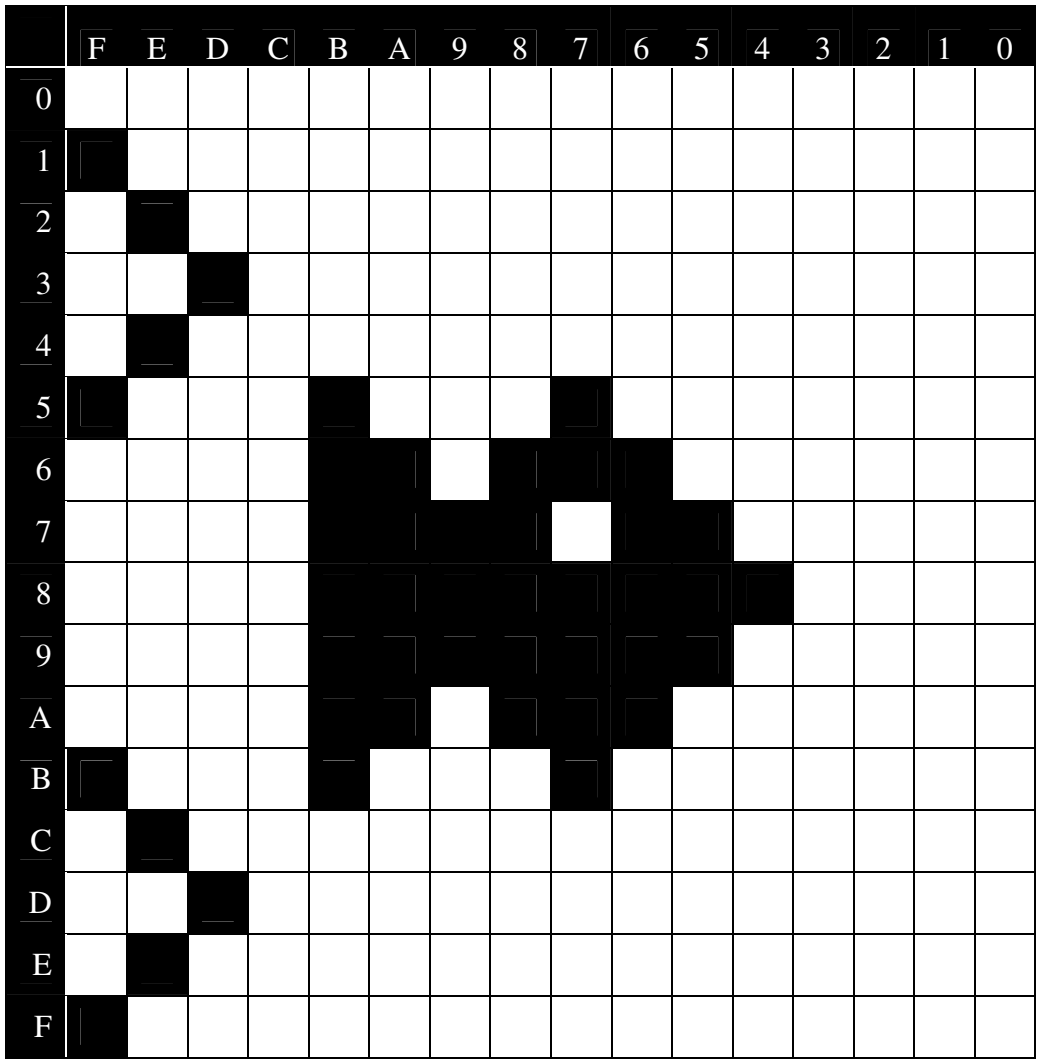
Frame 2



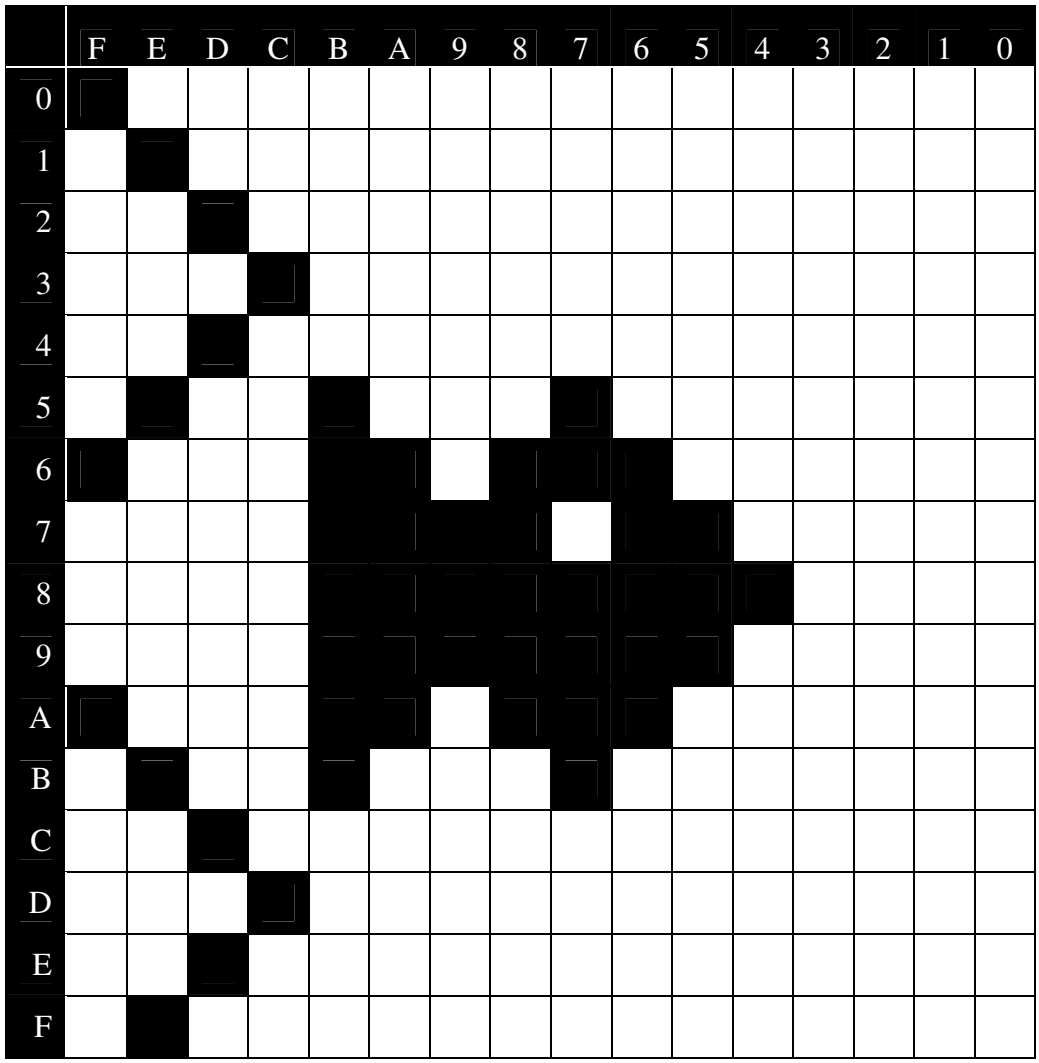
Frame 3



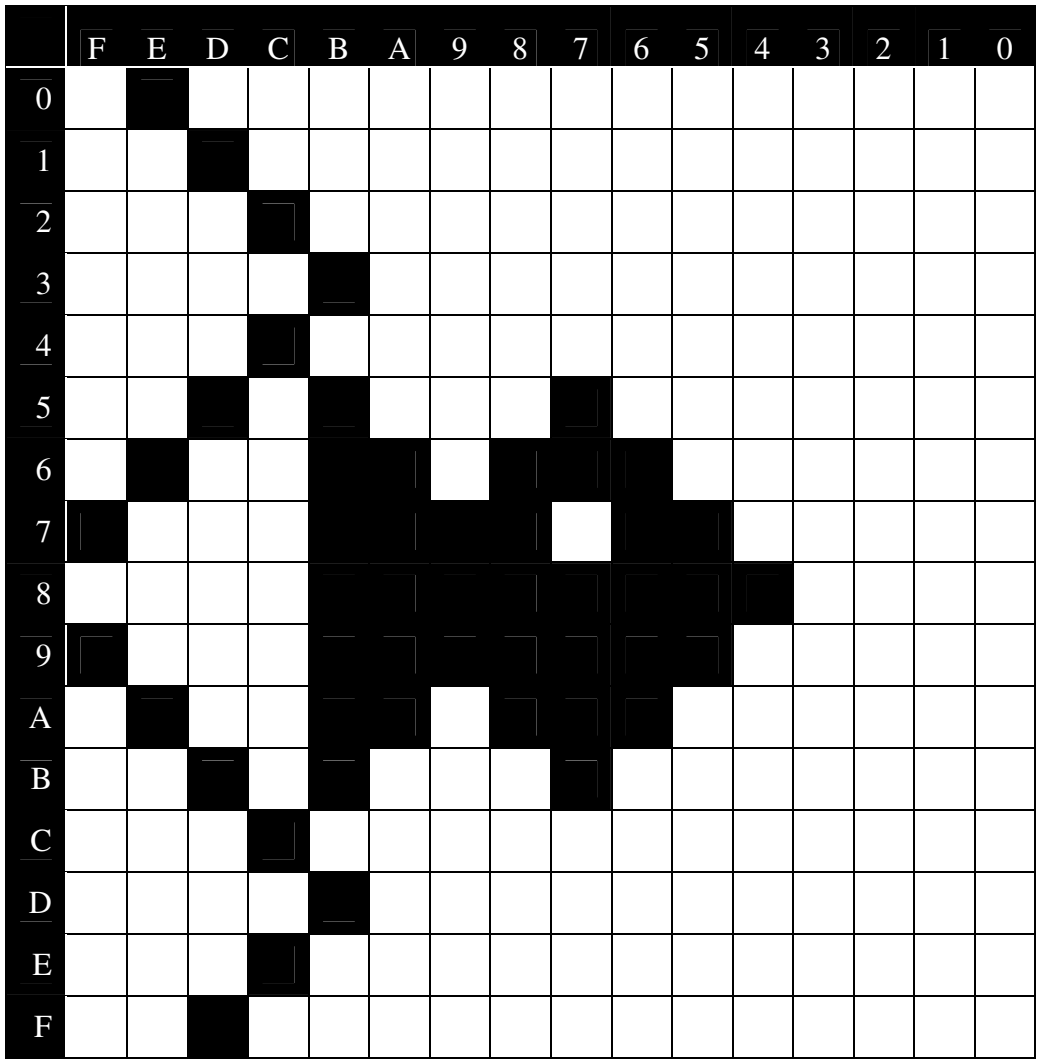
Frame 4



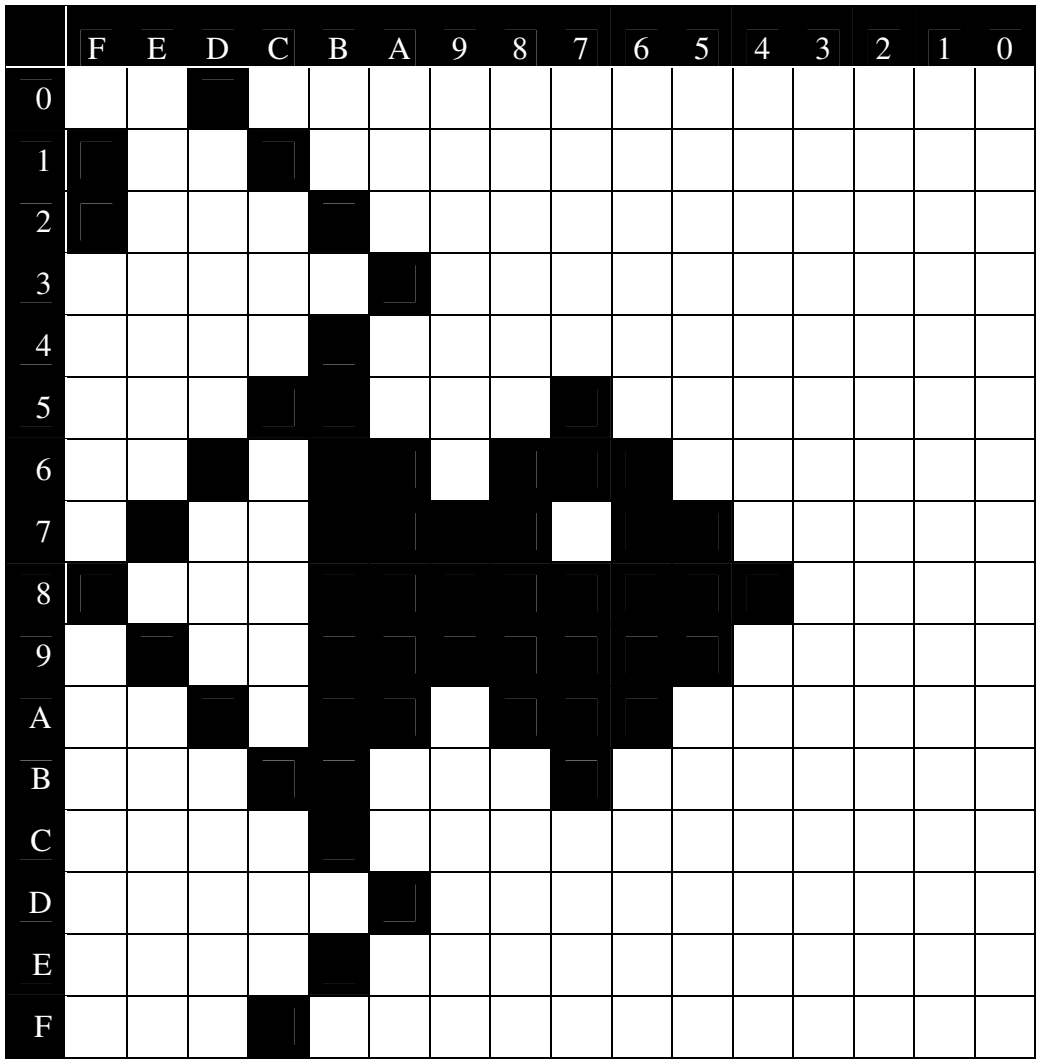
Frame 5



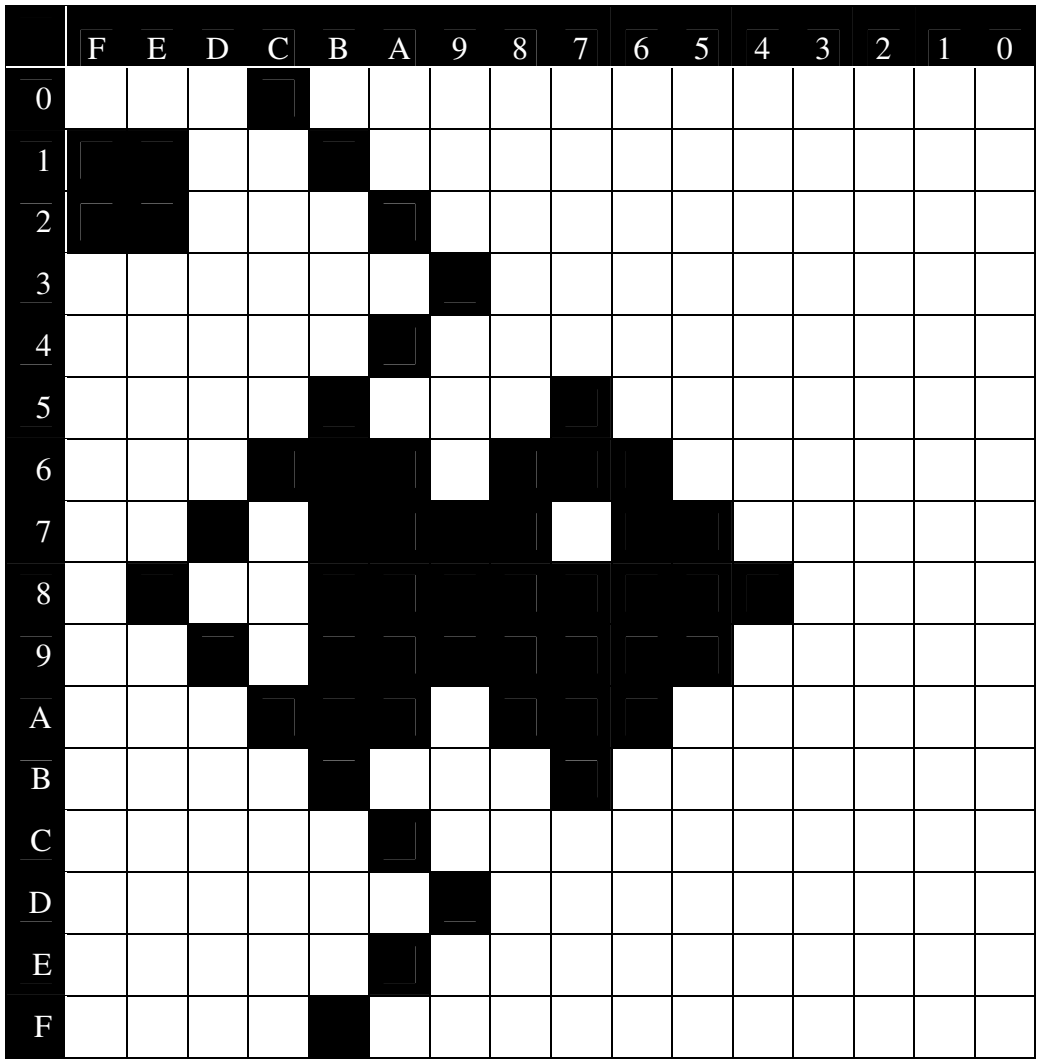
Frame 6



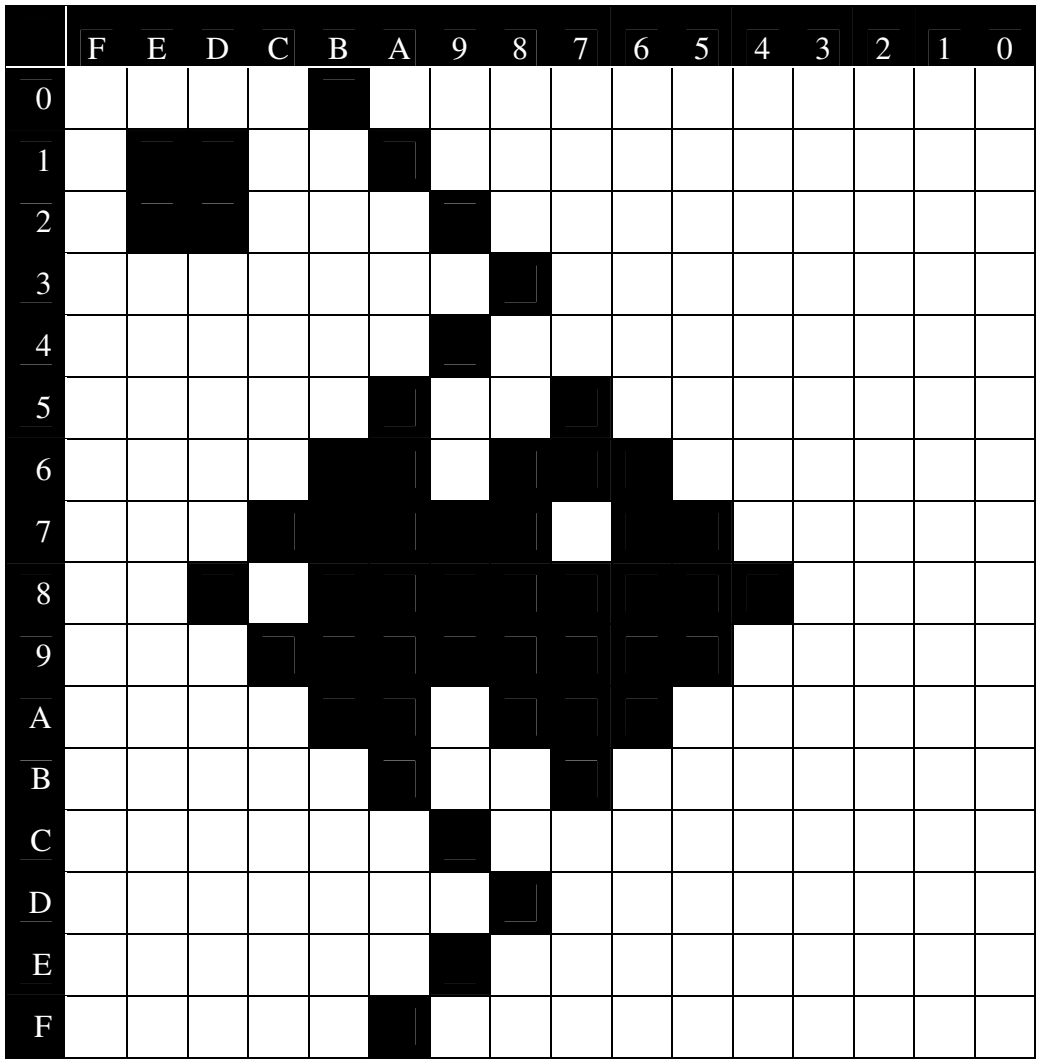
Frame 7



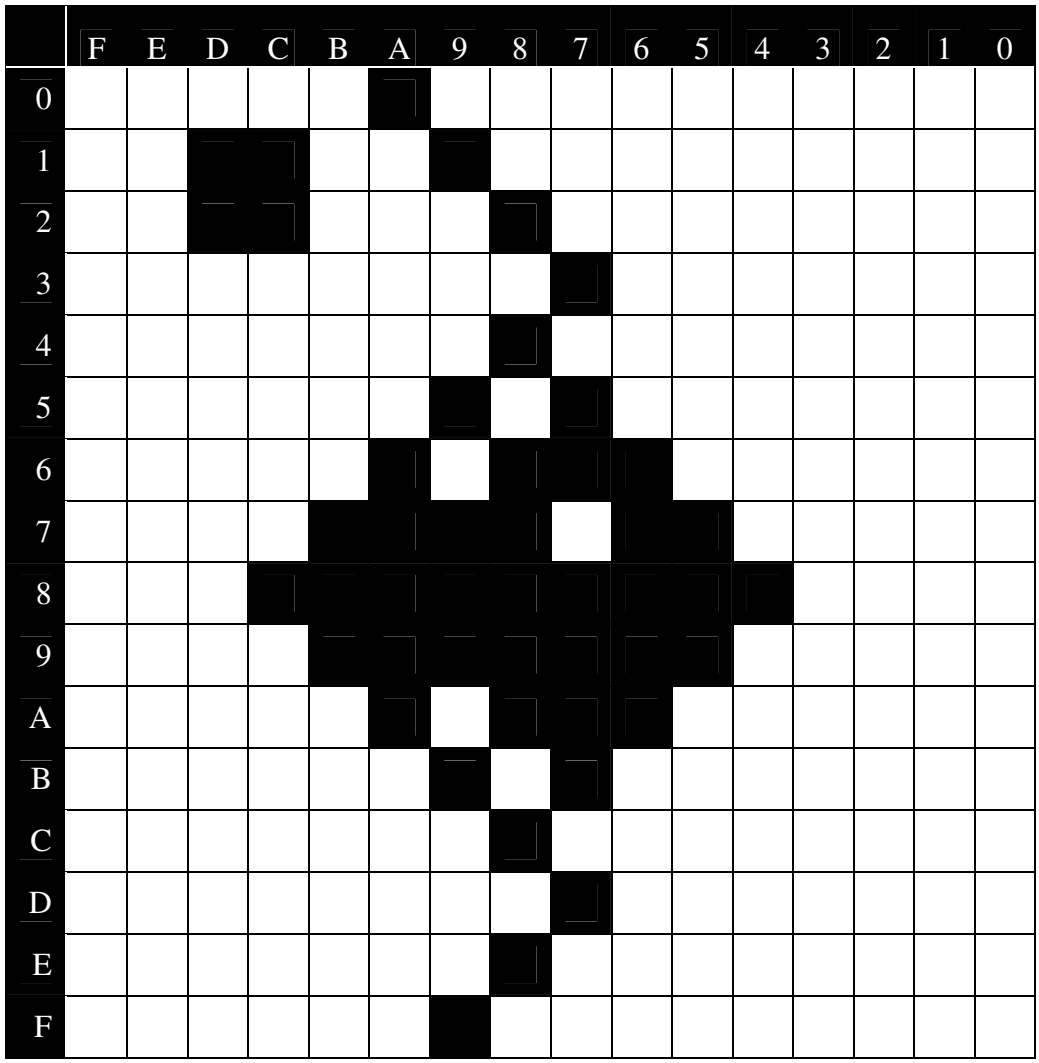
Frame 8



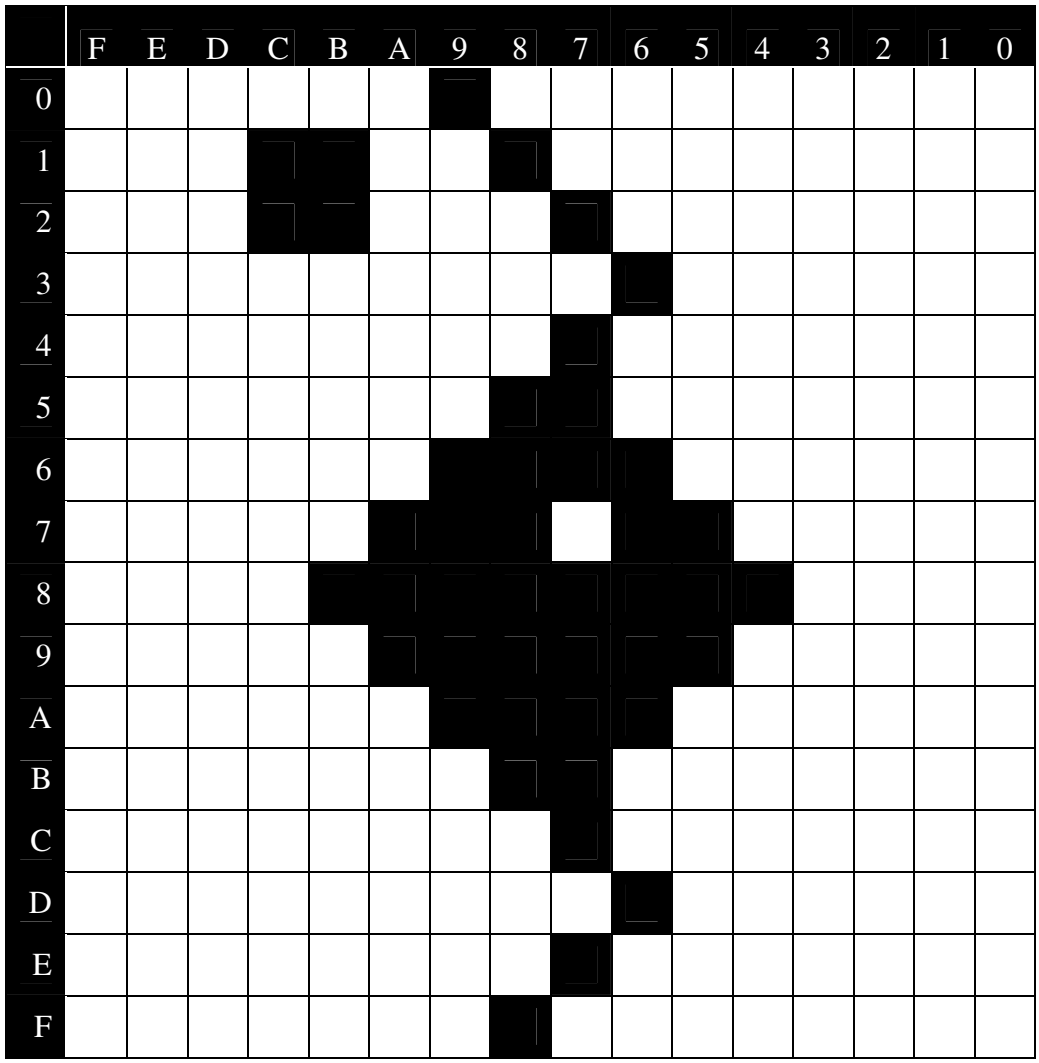
Frame 9



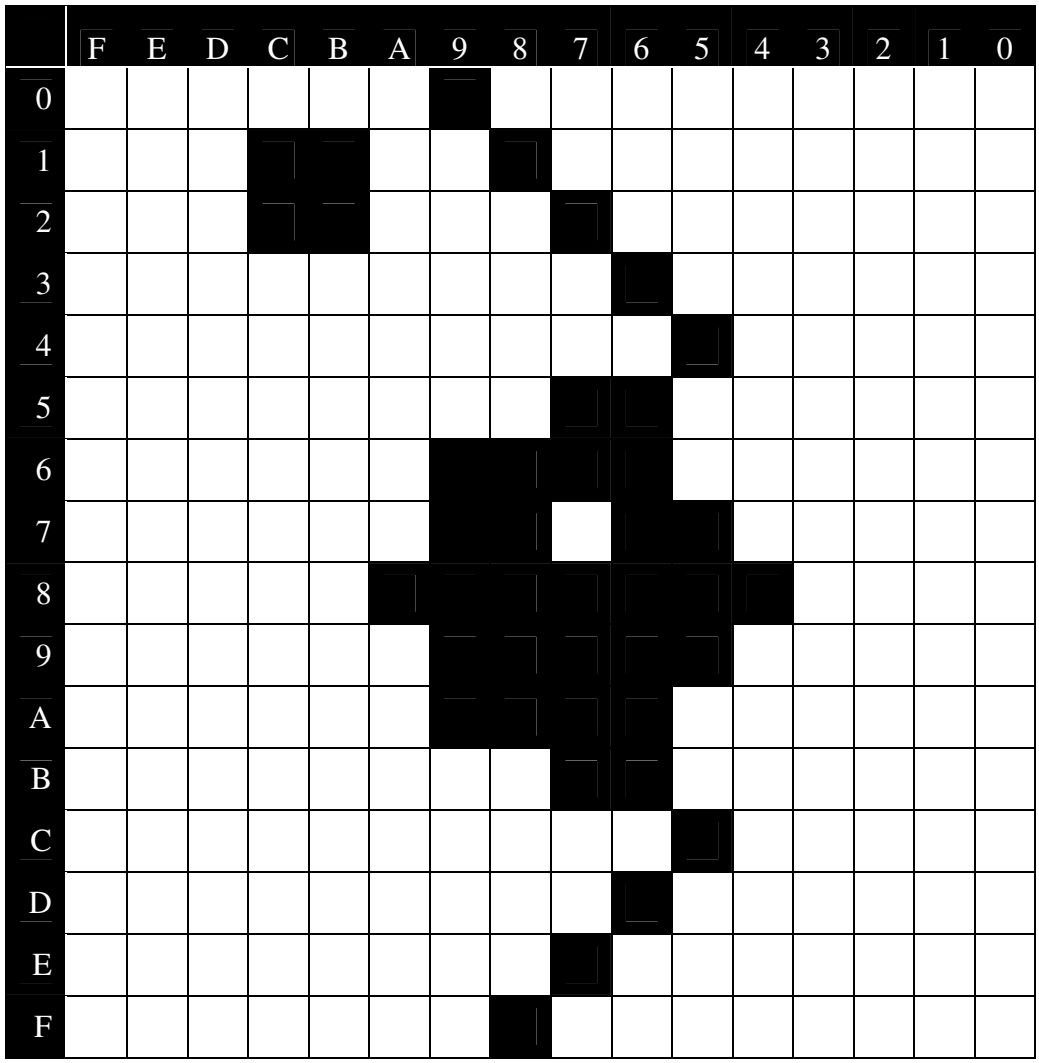
Frame 10



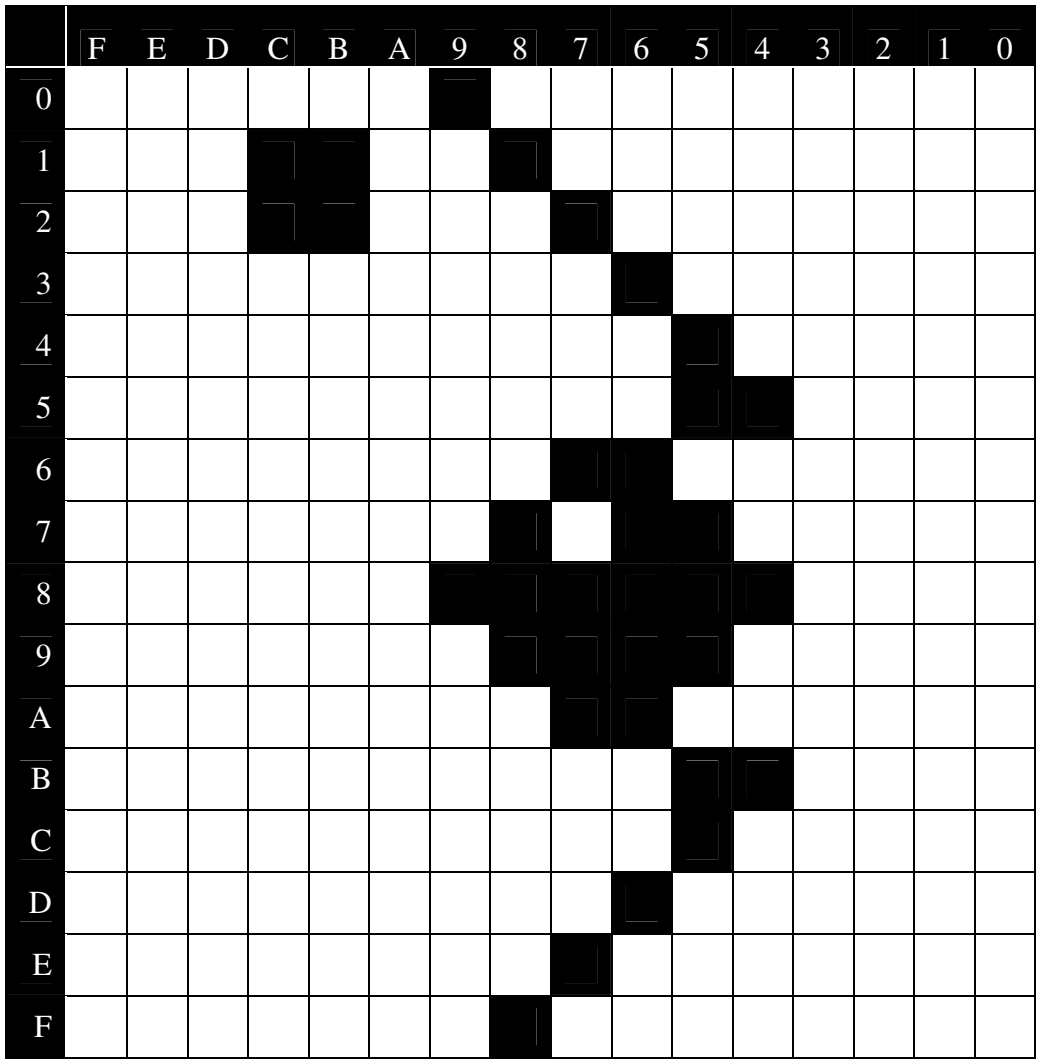
Frame 11



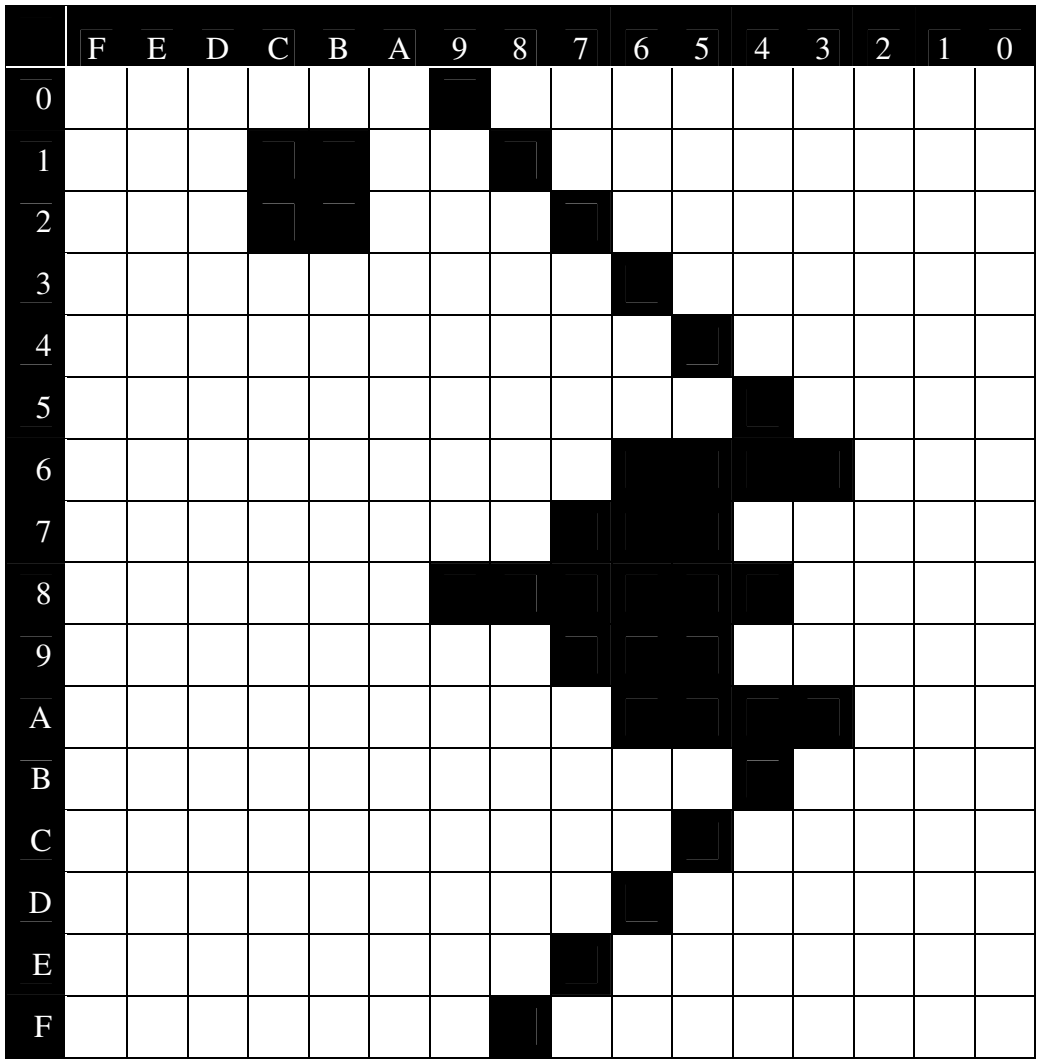
Frame 12



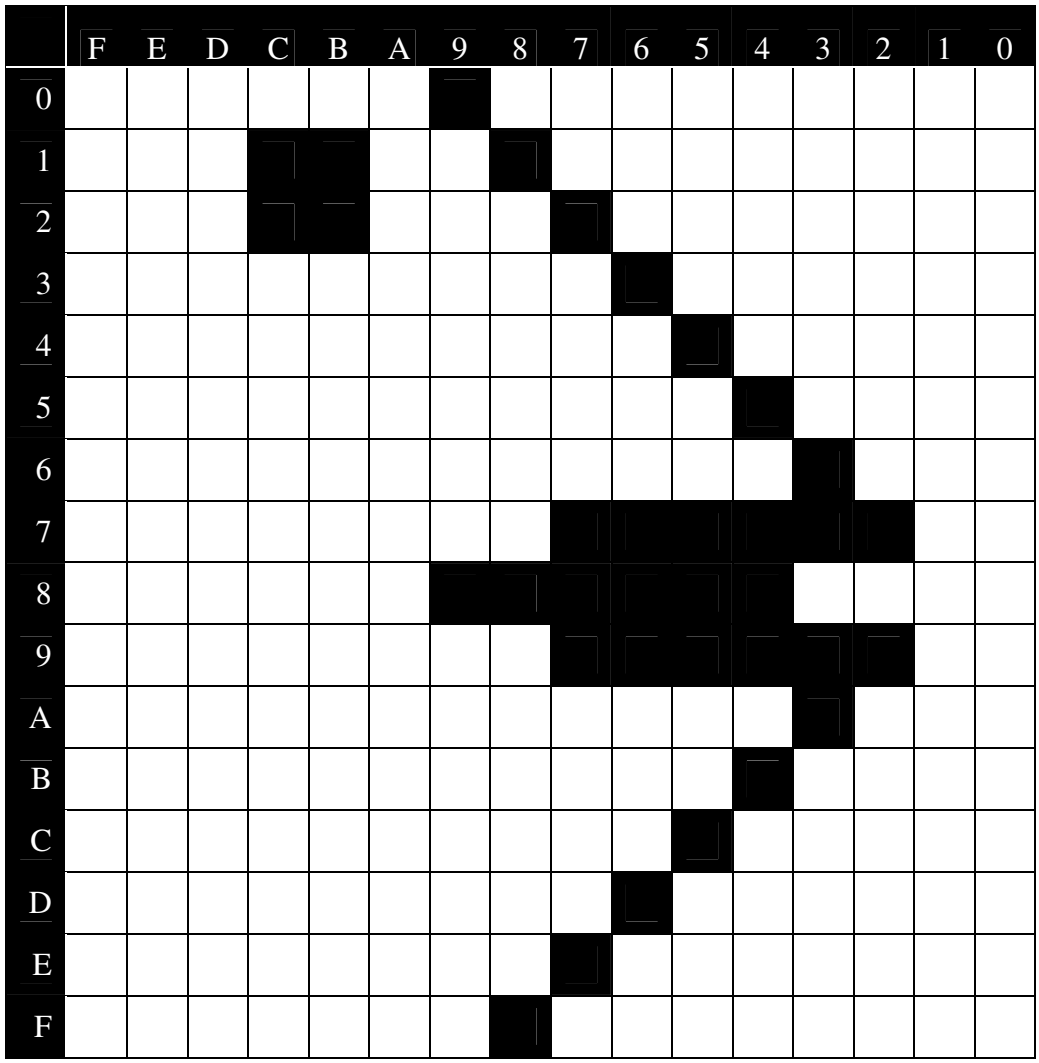
Frame 13



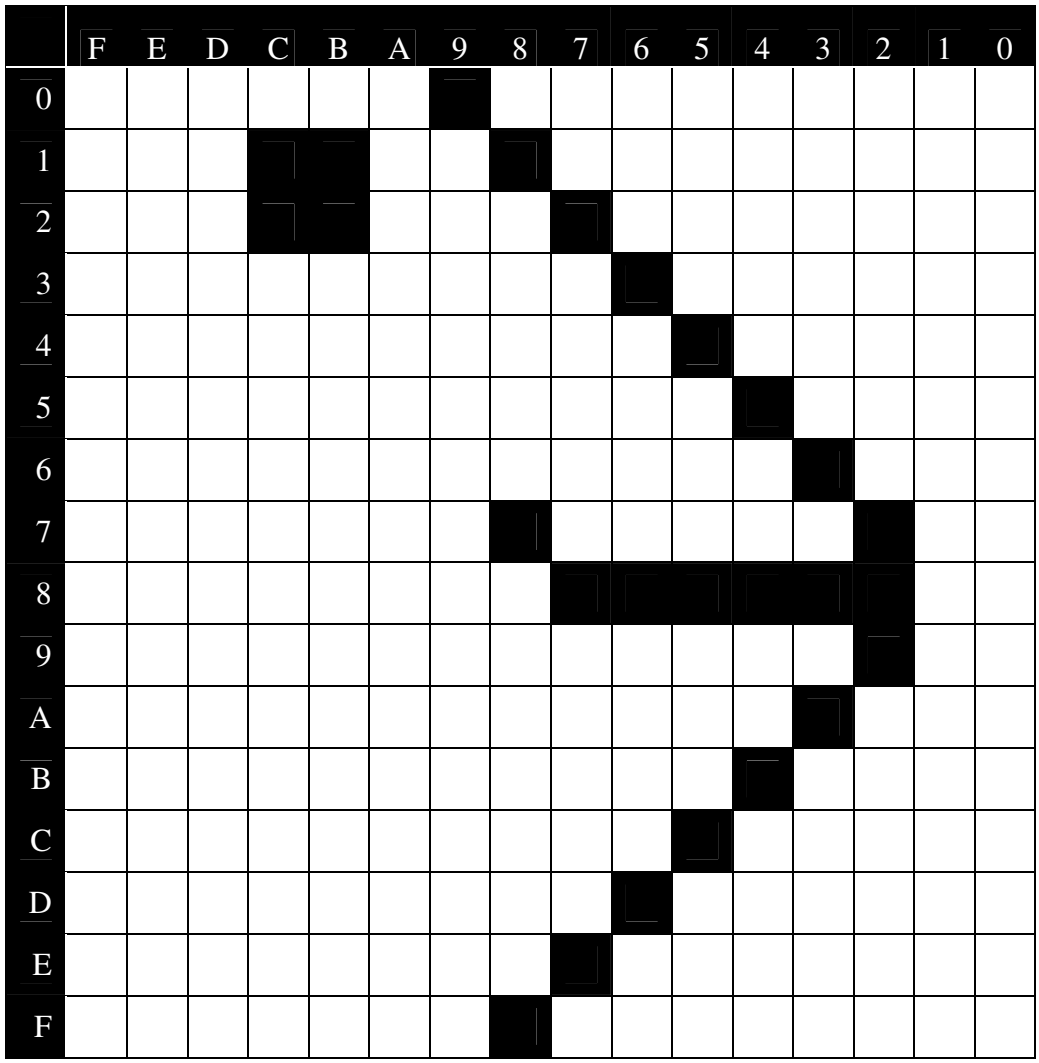
Frame 14



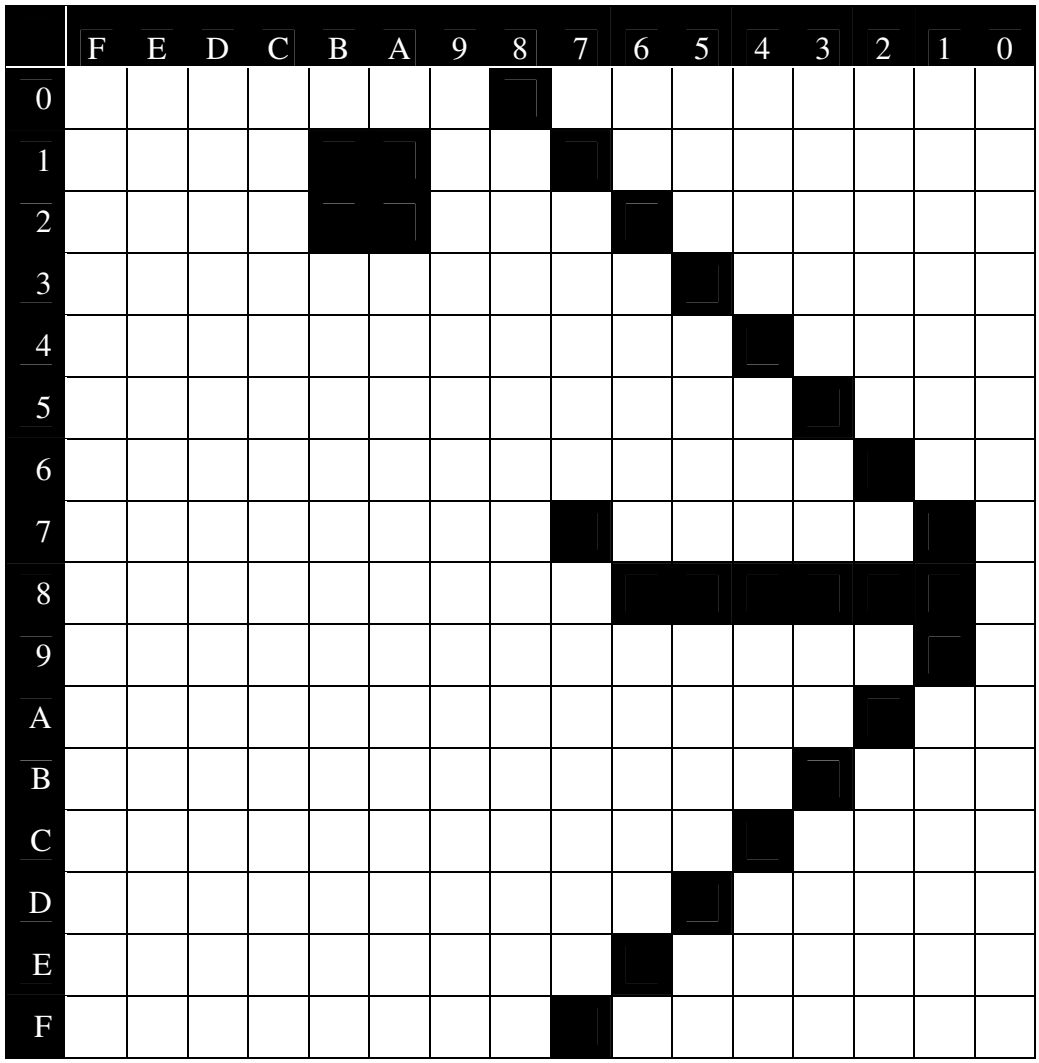
Frame 15



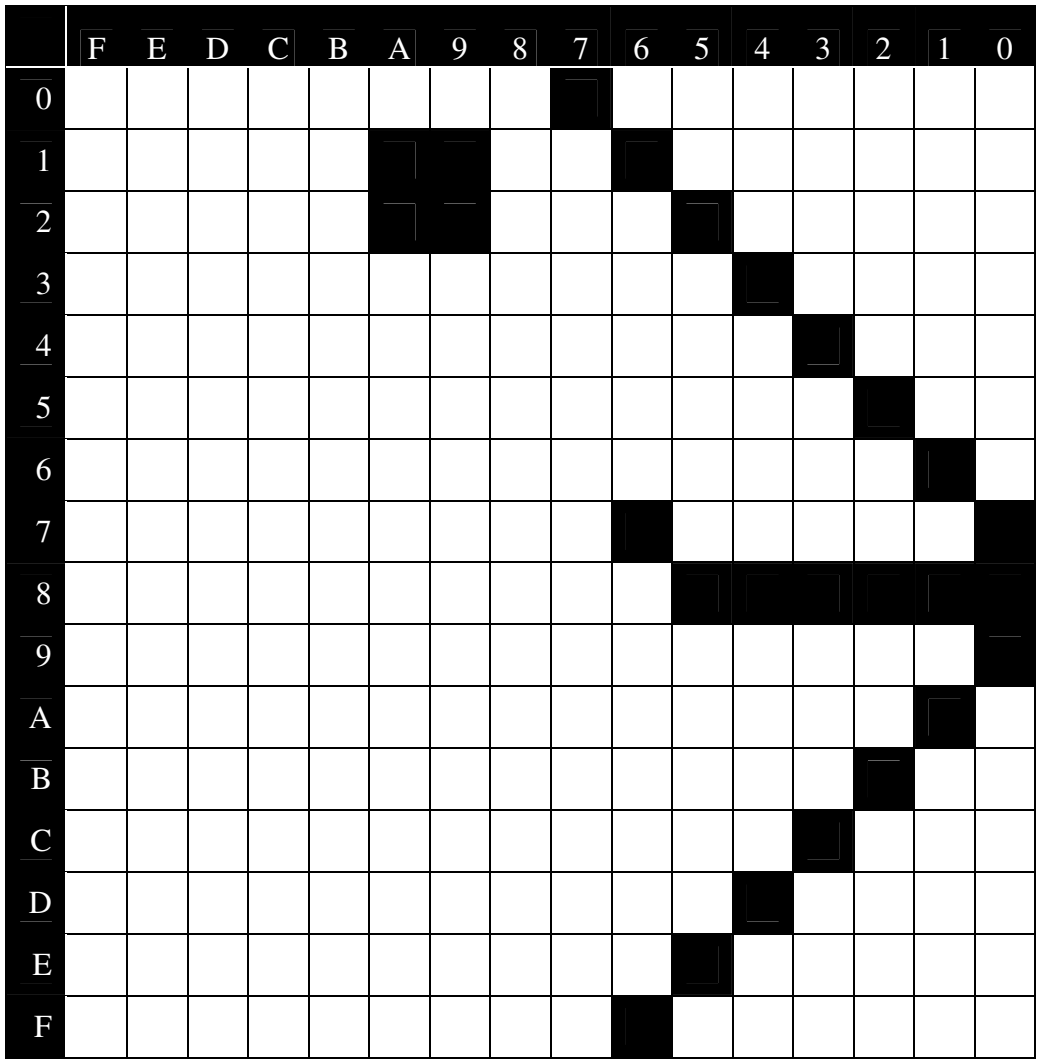
Frame 16



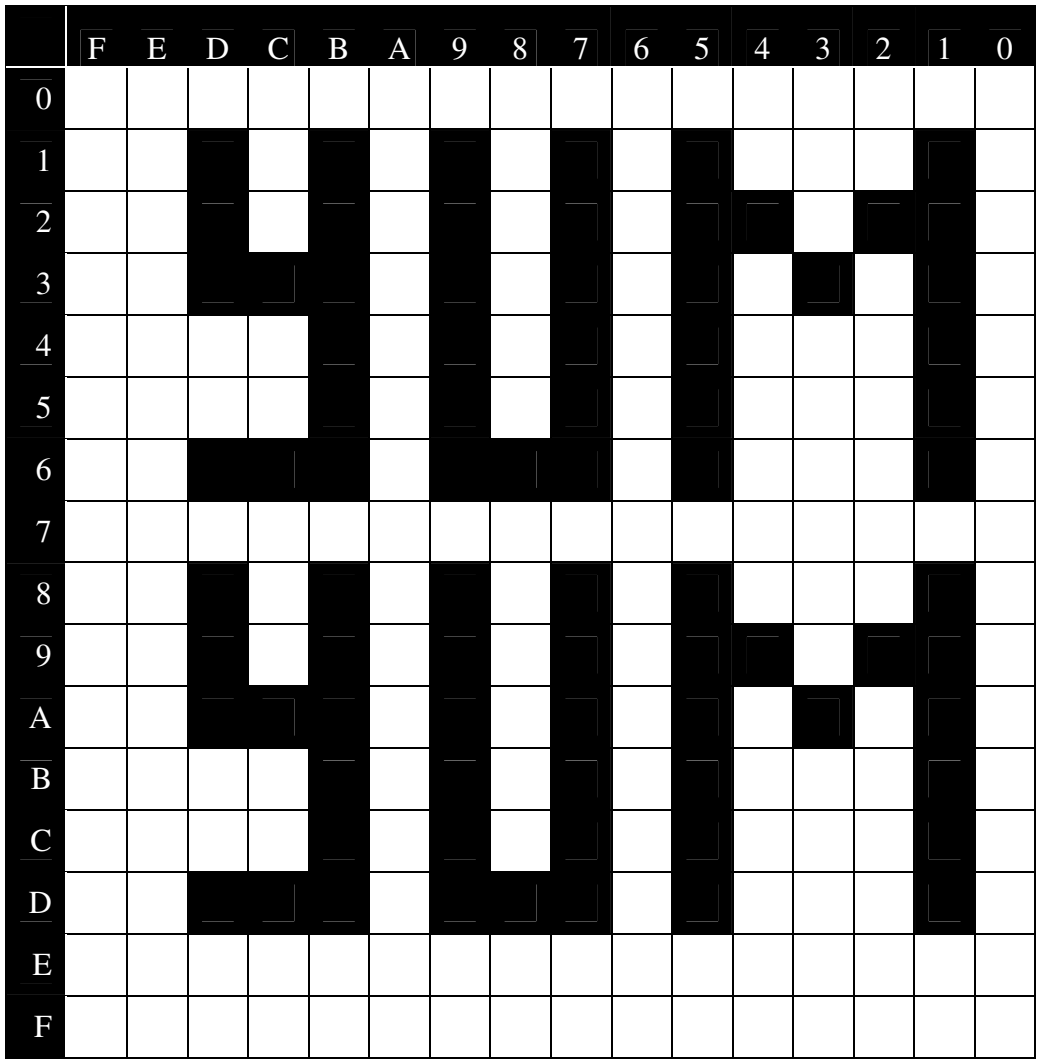
Frame 17



Frame 18



Frame 19



BIBLIOGRAPHY

- University of York Computer Science Department - Chips To Systems Module Website by Chris Bailey http://www-course.cs.york.ac.uk/cts
- DC Power Supply GPC-M Series Analogue Digital Type GW-Instek User Manual
- GOS – 6xxG Family Dual Trace Oscilloscope GW-Instek User Manual
- Zilog Z80 Family CPY User Manual UM0080020202
- The National Semiconductor Website www.national.com (for chip datasheets)
- Farnell InOne Website www.farnell.com (for chip costing)

Please note that some of the diagrams are taken directly from the national semiconductor website or Chris Bailey's CTS site or are modified versions of them.