Overview:
Harry Porter's
Relay Computer

I have built a computer out of relays. More info, including pictures:

http://www.cs.pdx.edu/~harry/Relay/index.html

The computer consists of 4 components:
- ALU (Arithmetic Logic Unit)
- Register Unit
- Program Control Unit
- Sequencing Unit

Each component is housed in a wooden cabinet and all wiring is visible behind the acrylic front panel. Switches and lights allow the operator to see and change the internal state.

Each cabinet is Mahogany and meant to hang on the wall or stand on a table. Each cabinet is approximately 28" x 40" x 8".

Features

- 8 general purpose registers (of 8 bits each)
- instruction register (8 bits)
- program counter (16 bits)
- 2 additional registers (16 bits each)
- 7 bit ALU (operations: AND, OR, XOR, NOT, SHL, ADD, INC)
- 16 bit increment unit
- 32 Kbytes of main memory (implemented using one CMOS chip)

Each instruction is 8 bits long. Some instructions (JUMP, CALL, BRANCH, SET-16) are followed by 16 additional bits of immediate data.

The instruction set

MOVE	register to register
CLEAR	set register to zero
LOAD	1 byte from memory to register
STORE	1 byte from register to memory
AND	8 bit, register to register
OR	8 bit, register to register
XOR	8 bit, register to register
NOT	8 bit, register to register
SHL	8 bit, register to register, shift left 1 bit
ADD	8 bit, register to register
INCREMENT	8 bit, register to register
INCR-XY	16 bit, XY register
GOTO	unconditional
CALL	return address is stored in XY register
RETURN	jump indirect through a register
BRANCH	conditional (beq, bne, blt, bcy)
SET-16	load 16-bit immediate value into register (The 2-byte value follows the instruction.)
SET-8	load 5-bit sign-extended immediate value into register (with sign extension)
HALT	suspend execution

The clock cycle time is approximately 5 Hz. Instructions take between 8 and 24 cycles. Obviously not fast, but lights blink and it makes noise.

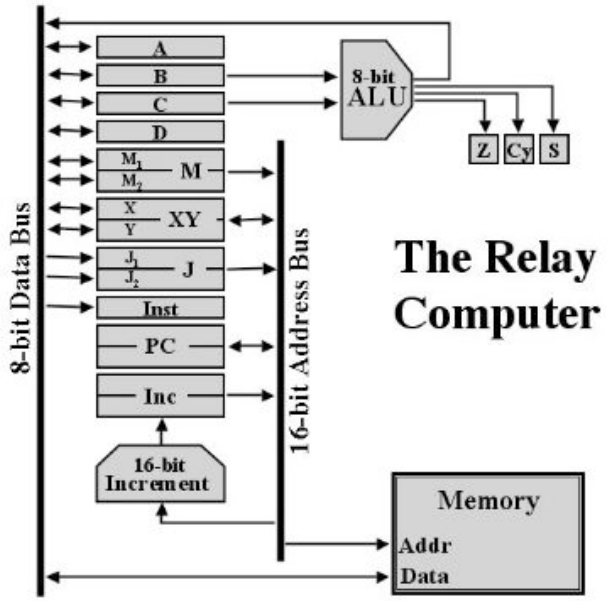
The Clock Circuit

The clock circuit has 4 relays and 4 capacitors. While the clock circuit has several internal states, its output is a single wire with a square wave. It's inputs are (1) the run/stop switch, (2) a line to freeze the clock, which is only activated when the HALT instruction is executed, and (4) a reset switch, for resuming after a HALT.

In more detail: the clock circuit involves 4 relays, connected in a chain, one after the other. Each relay has a capacitor (each is .5 farad). The relays can be numbered 1, 2, 3, 4. The chain is a cycle so relay 1 follows 4. At any one time, two relays are on. Call them relay i and i+1. Relay i is being kept on by its capacitor, which is discharging. Relay i+1 has a signal to it, so relay i+1 is charging up its capacitor. When the capacitor for relay i finally depletes and the relay switches off, this triggers a change in the circuit. It will shut off the signal to relay i+1 (which will then begin to discharge) and it will also turn on the signal to i+2, which will turn on and begin charging.

Registers and System Organization

Please consult the block diagram showing the overall system architecture.



The general purpose registers, which each have 8 bits, are called:

A, B, C, D, M1, M2, X, and Y

In some instructions, M1 and M2 are combined to form a 16-bit register, called M. Likewise, in some instructions, X and Y are combined to form a 16-bit register, called XY.

The program counter (PC) is 16-bits.

There is a 16 increment unit, which adds 1, and there is a 16-bit register called INC dedicated to the increment unit. This register is not visible to the programmer's model and is used only (1) to increment the PC during each instruction, and (2) in the INCR-16 instruction.

The instruction register (INST) is 8-bits and holds the instruction being executed.

There is a 16-bit register, called J, which is used during the CALL, JUMP, and GOTO instructions. These instructions load the register from the immediate 16-bit value following the instruction, and then to complete the transfer of control, the J register is copied to the PC register.

The computer is not pipelined. Each instruction is executed to completion before the next instruction is fetched.

There is a 3-bit condition code register, with the following bits:

- Z = set when value is zero
- Cy = set when there is a carry from the ALU for ADD or INCREMENT operations
- S = set when the value is negative, i.e., when most significant bit is 1

The condition code register is set after any ALU instruction. The ALU instructions are:
AND, OR, XOR, NOT, SHL, ADD, INCREMENT

There are 2 buses. The Data Bus is 8-bits wide and the Address Buss is 16-bits wide.

The Main Memory is connected to both the the address bus and the data bus.

The Main Memory

The Main Memory Unit is organized as 32K bytes, addressed from hex 0000 through 7FFF. It is implemented with a single CMOS Static RAM chip, Hitachi part HM62256 Series.

While all the realys are driven by the large 12 Volt power supply, there is a small 5 Volt power supply used only to power the RAM chip. A set of 8 of FET amplifiers are used to change from the chip's TTL levels to 12 Volts to drive the relays, which is needed for transferring data from the memory to the CPU in a "memory-read" operation. There are also three small LED arrays, each containing 10 LEDs that run at TTL levels. These are used to monitor the address lines (15) and data lines (8) right at the chip.

The MOV Instruction

The MOV instruction "selects" one register, whose contents are then dumped (i.e., gated) onto the Data Bus. Then, the destination register is "loaded" from the bus. Any general purpose register can be the source and any general purpose register can be the destination.

The MOV instruction is encoded as follows:

```
0 0 d d d s s s
```

where "ddd" and "sss" specify the destination and source registers according to this encoding:

```
0 0 0 = A
0 0 1 = B
0 1 0 = C
0 1 1 = D
1 0 0 = M1
1 0 1 = M2
1 1 0 = X
1 1 1 = Y
```

The CLEAR Instruction

This instructuion moves 0 into one of the 8-bit registers (A, B, C, D, M1, M2, X, or Y).

This instruction is encoded as a variation of the MOV instruction, where "sss" and "ddd" are the same.

The LOAD and STORE Instructions

The memory is organized as 32K bytes, addressed from hex 0000 through 7FFF.

The LOAD instruction uses the 16-bit value in the M register as an address and reads a byte from Main Memory into either the A, B, C, or D registers. Likewise, the STORE instruction assumes the address is in the M register and stores the value from either A, B, C, or D into a byte in Main Memory.

The LOAD instruction is encoded as follows:

```
1 0 0 1 0 x r r
```

where "x" is ignored and "rr" specifies the destination register according to this encoding:

```
0 0 = A
0 1 = B
```

```
1 0 = C
1 1 = D
```

The STORE instruction is encoded as follows:

```
1 0 0 1 1 x r r
```

where "x" is ignored and "rr" specifies the source register according to this encoding:

```
0 0 = A
0 1 = B
1 0 = C
1 1 = D
```

The ALU Instructions

The ALU takes as inputs the current values of registers B and C. Its result is placed on the 8-bit bus. In theory, the result can be transfered into any general purpose register, except B or C, which would cause feedback. However, the ALU instructions can place the result into either A or D. This class of instructions can perform these operations:

```
A = B + C     D = B + C      8-bit addition
A = B + 1     D = B + 1      8-bit increment
A = B & C     D = B & C      logical AND
A = B | C     D = B | C      logical OR
A = B xor C   D = B xor C    logical EXCLUSIVE-OR
A = !B        D = !B         logical NOT
A = B << 1    D = B << 1     left-shift circular
```

The 3-bit condition code register is set by this instruction, according to the result:

```
S    "Sign bit"    Set if result is negative, i.e., iff most sign. bit is "1".
Cy   "Carry bit"   Set if there was a carry-out for ADD or INCREMENT.
Z    "Zero bit"    Set if result is "00000000".
```

The ALU instructions are encoded as follows:

```
1 0 0 0 r f f f
```

where "r" specifies the destination register according to this encoding:

```
0 = A
1 = D
```

And "fff" specifies the operation according to this encoding:

```
0 0 0 = B + C
0 0 1 = B + 1
0 1 0 = B & C
0 1 1 = B | C
1 0 0 = B xor C
1 0 1 = !B
1 1 0 = B << 1
```

The INCR-XY Instruction

This instruction adds 1 to the 16-bit XY register. It is encoded as:

```
1 0 1 1 0 0 0 0
```

The condition code register is not updated.

The GOTO Instruction

The GOTO instruction is 3 bytes long. The 2nd and 3rd bytes contain a 16-bit address. A jump to this address is made. This instruction is encoded as:

```
1 1 1 0 0 1 1 0   a a a a a a a a   a a a a a a a a
```

where the field "aaaaaaaa aaaaaaaa" is the target address.

The CALL Instruction

The CALL instruction is exactly like the GOTO instruction, except that, in addition, the address of the next instruction after the CALL instruction is saved in the XY register.

This instruction is encoded as:

```
1 1 1 0 0 1 1 1   a a a a a a a a   a a a a a a a a
```

where the field "aaaaaaaa aaaaaaaa" is the address of the subroutine.

The RETURN Instruction

The RETURN instruction moves the contents of XY back to the PC. This can be used to effect a RETURN (when used after a CALL) or an indirect jump, when XY contains a computed value.

This instruction is slightly more flexible and is actually a 16-bit move instruction that can perform any one of the following moves:

```
PC = XY
PC = M
PC = J
XY = M
XY = J
XY = 0
```

This instruction is encoded as:

```
1 0 1 0 d s s 0
```

The field "d" specifies the destination register, according to this encoding:

```
0 = XY
1 = PC
```

The field "ss" specifies the source register, according to this encoding:

```
0 0 = M
0 1 = XY
1 0 = J
1 1 = 0 and halt the machine (See the HALT instruction)
```

The Conditional BRANCH Instructions

The conditional BRANCH is 3 bytes long. The 2nd and 3rd bytes contain the 16-bit target address. The condition BRANCH instruction will test the condition code register and optionally jump to the target address. Here are different tests that can be performed:

```
BE   Take the branch if Z=1, i.e., if the last result was zero
BNE  Take the branch if Z=0, i.e., if the last result was not zero
BNC  Take the branch if Cy=0, i.e., if there was no carry for the ADD or INCR
BNEG Take the branch if S=1, i.e., if the last result was negative
```

These can also be combined, in which case the branch is taken if any of the conditions is met. (The unconditional GOTO is actually a conditional branch testing for "Z=1 OR Z=0", which is always true.)

Note that "BE" and "BZ" are the same. Also "BNE" and "BNZ" are the same. Which is preferred depends on the context in which the instruction is used. "BE" would be preferred after a XOR instruction to compare 2 values for equality:

```
     A = B XOR C
     BE target
     < execution reaches here if B and C are different >
```

This instruction is encoded as:

```
     BNEG = 1 1 1 1 0 0 0 0    a a a a a a a a    a a a a a a a a
     BNC =  1 1 1 0 1 0 0 0    a a a a a a a a    a a a a a a a a
     BE =   1 1 1 0 0 1 0 0    a a a a a a a a    a a a a a a a a
     BNE =  1 1 1 0 0 0 1 0    a a a a a a a a    a a a a a a a a
```

where the field "aaaaaaaa aaaaaaaa" is the target address.

The SET-16 Instruction

The SET-16 instruction is 3 bytes long. The 2nd and 3rd bytes contain a 16-bit value. This value is loaded into the M register.

This instruction is encoded as:

```
     1 1 0 0 0 0 0 0    v v v v v v v v    v v v v v v v v
```

where the field "vvvvvvvv vvvvvvvv" is the value to load into the M1/M2 register pair.

The GOTO/CALL/BRANCH/SET-16 Instruction Class

These instructions are all really variations of a single instruction. This super-instruction has the following encoding:

```
     1 1 r s c z n x    a a a a a a a a    a a a a a a a a
```

The bit field in the op-code have the following meanings:

```
     r: Load the 2nd and 3rd bytes of the instruction into
      either the M or J registers. (r=0 means load M; r=1 means load J)
     x: When set, copy the PC into XY before jumping (used for CALL)
     s: When set and when S=1, copy register J to PC (take the branch)
     c: When set and when Cy=0, copy register J to PC (take the branch)
     z: When set and when Z=1, copy register J to PC (take the branch)
     n: When set and when Z=0, copy register J to PC (take the branch)
```

Thus it is possible to load the J register without jumping, which can be used to load the XY register. (The instruction would be followed by a variant of the "return" instruction which performed a "XY=J" move.) Also, the condition branching can be combined with saving the PC in XY, resulting in a "CONDITIONAL CALL" instruction.

The SET-8 Instruction

This instruction can be used to load register A or B with any value in the range:

```
     -16 .. +15
```

This instruction is encoded as follows:

```
     0 1 r v v v v v
```

The 5-bit "vvvvv" field is sign-extended to 8 bits. The "r" bit is used to select the destination register, according to the following encoding:

```
     0 = A
     1 = B
```

The HALT Instruction

This instruction will suspend execution by freezing the clock. Instruction execution is resumed by toggling a RESET switch. This instruction will also clear the PC. It is encoded as:

    1 0 1 0 1 1 1 0

Timeline, History, and General Comments

I teach CS and I have always loved computers and been interested in how they work. Over the years I have found that you can read and study and make paper designs, but there is no substitute for actually building things. There are always processes at work that you can't understand until you actually build a working unit. I had wanted to build a computer out of relays when I was much younger, but I didn't have the knowledge, patience, money, or time.

A couple of years ago, at the technical museum in Munich, I saw a replica of Konrad Zuse's Z3 computer and I copied down his circuit diagram for the fuller adder, which was quite clever. Later, as a little project, I built a simple adder, using this circuit and really enjoyed the physicality of it, although it could only add two 2-bit numbers. I then started thinking about constructing a full 8-bit ALU. After I had completed the basic design, I couldn't help thinking about building an entire computer.

Giving final exams are a panic for students, but are really different for the professors. It is the only time that there are no lectures to plan and no exams to grade. As I sat there during a final one term, I sketched out the outline of an architecture that I might use. So I designed the ALU to "fit" into a complete machine. However, when I decided to build the ALU, I still wasn't committed to building anything more. It is important to stay focussed on a project that you can complete. It is easy (and fun) to fantasize about building something big, but unless one is able to concentrate long enough to get something completed, it is really just idle day-dreaming!

Once I finished the ALU, I decided to go ahead and build the second cabinet, the register unit. However, in my mind, I had not committed to building anything beyond that. I figured that if I finished it and it worked, I would then decide whether I wanted to proceed. I knew there was a possibility I might be burned out, or might run into problems that would make a full computer non-functional. I always want successes, not failures, so I decided I would rather have a fully-working half-computer that a half-finished full-computer.

After finishing the ALU and the register unit, I found I enjoyed all aspects of the construction. Usually design is the most fun and, when building stuff, we tend to leave the most unpleasant or difficult tasks until the very end, but after completing 2 cabinets, I knew pretty much what would be involved. At that time I decided to keep going.

All in all, it took over a year, but I didn't work on it very often. However, projects that are fun always go faster and seem to take less time!!! You squeeze in the time while watching TV, you go over the circuits in your head while going to sleep, and you never forget to stop at the hardware store when you happen to be in the area! Part of the secret to being able to do big projects is your own personal motivation.

I was not on a budget, so this helped.

Costs

415 Relays, 4PRLY-12, plus extras for prototyping ($3.40 each)
350 LEDs ($1.49 each)
111 Switches, SPDT, on-on mini toggle, MTS-4 ($0.90 each)
Acrylic boards, pre-cut and pre-drilled, 4 cabinets (total $1,095.37, aprx $275 per cabinet)
    ALU (1 board), 82.19
    Register Cabinet (10 boards), $251.60 total
    PGM-CTRL Cabinet (10 boards), $381.80 total
    Sequencer Cabinet (5 boards): $379.78 total
4 Mahogany Cabinets (est $300 each)
2 Power Supplies, 12V, 10Amp ($79.99 each)
20 Capacitors, 100UF, 100V non-polar ($1.55 each)
Memory Board:
    1 SRAM, 32K dip, Jameco # 82472CA, Other #: 62256LP-70 ($5.49)
    1 eight-channel FET driver module, NCD, www.controlanything.com, IOTEST-L ($49.00)
    3 eight-channel LED array module, NCD, www.controlanything.com, 8-FET ($10 each)
    1 Prototyping board ($5.99)
    1 small power supply (for memory) 5V, 4Amp, 20Watt, Jameco #: 213583CA, ($26.95)

DB-9 Sub-miniature Connectors
     32 Plugs ($1.59 each)
     32 Receptacles ($1.76 each)
     64 Connector hoods ($0.49 each)
22-Gauge black solid copper wire, 100 feet ($4.49 each), est. 20 rolls
8-connector cable, CAT5e, 4 pairs, 24AWG solid, 328 feet ($42.00)
4 Fans ($10 each)
Misc Hardware. (Est $100)

Summary:
     Relays, $1,411
     LEDs, $521
     Switches, $100
     Acrylic Boards, $1,095
     Cabinets, $1,200
     Power Supplies $160
     SRAM Memory, $117
     Capacitors, $148
     Connectors, $138
     Wire, $132
     Fans, $20
     Hardware, $100

Grand Total, $5,142 (per unit: $1,285)

Operation of the Machine

There are no input or output instructions. You use the computer by loading memory with a program. This is done through the switches, as follows:

     REPEAT
          Set 16 bit address switches to the memory address of the next instruction.
          Set 8 data switches to the bits of the instruction.
          Toggle the "memory write" switch.
     UNTIL the entire program has been entered.

You can put the input data into registers bit-by-bit by toggling the switches.

To execute the program:

     Initialize the "Program Counter" register to the address of the first instruction.
     Toggle a "Reset" switch.
     Flip the "Run-Stop" switch to "Run".

There is a HALT instruction which, when encountered, while cause the machine to freeze up. So far, all programs leave their results in registers, and you can read the contents of the registers (in binary) by looking at the lights.