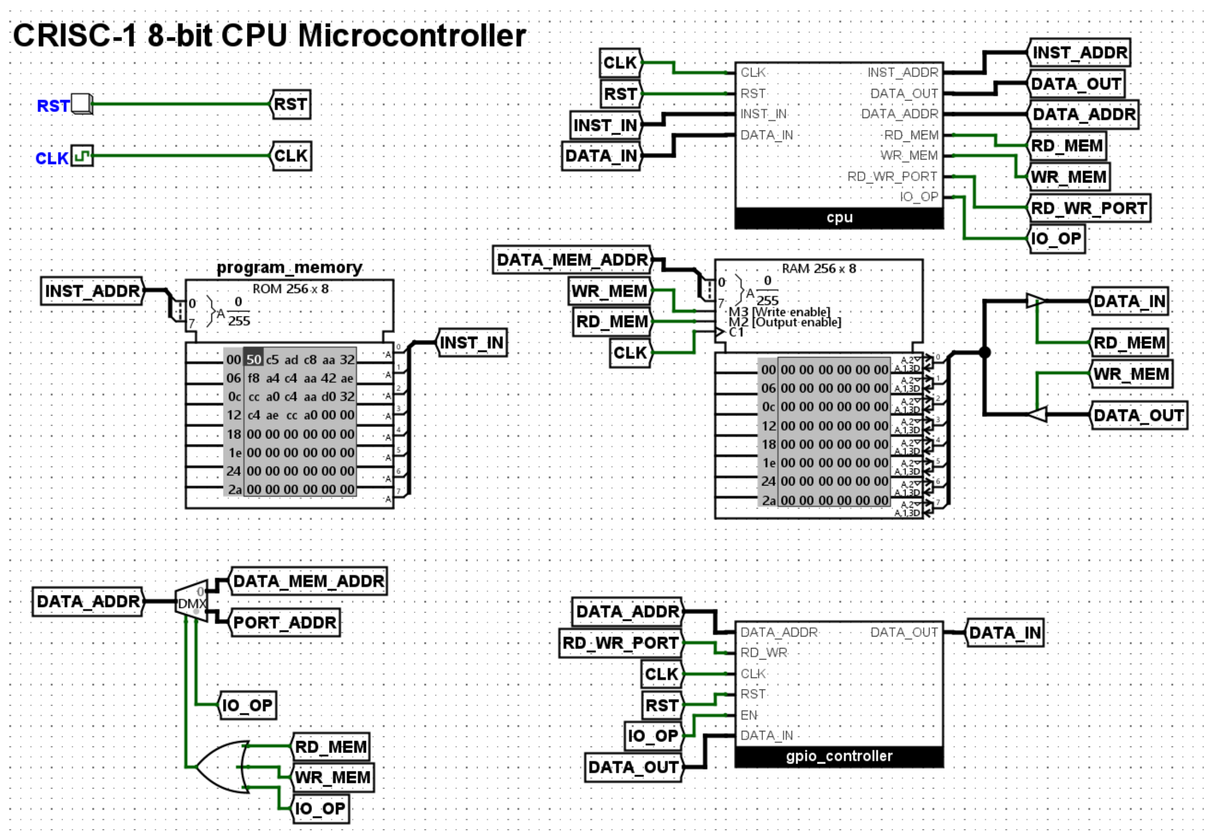
CRISC-1 Reference Manual

Pedro Botelho - 02/2024

1) About the CPU

The CRISC CPU , or *Crabby RISC*, is a simplistic 8-bit RISC-based processor, made to learn and teach computer science subjects, such as “Computer Architecture”, “Digital Logic” and “Assembly Language”.



It has some major characteristics, such as:

- Four 8-bit general-purpose registers (R0 to R3).
- Harvard architecture (not modified) with separated I/O addressing space.
- 8-bit address bus width, being able to operate with up to 256 X 8-bit memories.
- 19 single-cycle simple instructions with up to two operands (destination and source).
- Conditional branching based on a zero flag generated by the ALU.
- 16 I/O software-controller registers, which enables up to five software-controlled GPIO ports (PORT A to PORT E), with 8 pins each (P0 to P7) and three registers for user interface.

2) Instruction Set

The CPU has a rather small instruction set that takes advantage of the 8-bit instruction word, having a leading opcode that identifies the instruction group, some optional instruction specific opcode in the middle and the trailing register(s) identifier(s).

The instructions of an Assembly language made for this processor would typically have one of this formats:

- **No operand**: <INST>
- **One operand**: <INST> <RR>
- **Two operands (immediate)**: <INST> <RR>, <IMM>
- **Two operands (registers)**: <INST> <R1>, <R2>

The available instructions are listed below:

- **Control-specific**: Format (00000000).
  - Send a specific control signal to the processor.
  - **HALT**: Halts the processor.
    - **Symbol**: HALT
    - **Description**: Stops CPU processing signal.
    - **Binary code**: (00000000)

- **Logic-Arithmetic**:
  - Uses the Arithmetic and Logical Unit (ALU) to process data in the registers.
  - **Binary Operations**: Format (0)XXX[SSDD].
    - ALU operations that require two registers.
    - **ADD**: Adds two registers in the destination register.
      - **Symbol**: ADD DD, SS
      - **Logic**: DD = DD + SS
      - **Binary code**: (0)001[SSDD]
    - **SUB**: Subtracts two registers in the destination register.
      - **Symbol**: AND DD, SS
      - **Logic**: DD = DD & SS
      - **Binary code**: (0)010[SSDD]
    - **AND**: Does the AND operation between two registers and saves the result in the destination register.
      - **Symbol**: AND DD, SS
      - **Logic**: DD = DD & SS
      - **Binary code**: (0)011[SSDD]
    - **OR**: Does the OR operation between two registers and saves the result in the destination register.
      - **Symbol**: OR DD, SS
      - **Logic**: DD = DD | SS
      - **Binary code**: (0)100[SSDD]
    - **XOR**: Does the exclusive OR operation between two registers and saves the result in the destination register.

        - **Symbol**: XOR DD, SS
        - **Logic**: DD = DD ^ SS
        - **Binary code**: (0)101[SSDD]
    - **MUL**: Multiply two registers in the destination register.
        - **Symbol**: MUL DD, SS
        - **Logic**: DD = DD * SS
        - **Binary code**: (0)110[SSDD]
    - **CMP**: Compares two registers (doesn't write back to destination register).
        - **Symbol**: CMP DD, SS
        - **Logic**: ZF = DD - SS
        - **Binary code**: (0)111[SSDD]
  - **Unary Operations**: Format (1011)XX[DD].
    - ALU operations that require one register.
    - **NOT**: Inverts DD.
        - **Symbol**: NOT DD
        - **Logic**: DD = !DD
        - **Binary code**: (1011)00[DD]
    - **NEG**: Negates DD.
        - **Symbol**: NEG DD
        - **Logic**: DD = -DD
        - **Binary code**: (1011)01[DD]
    - **INC**: Increments DD by one.
        - **Symbol**: INC DD
        - **Logic**: DD = DD + 1
        - **Binary code**: (1011)10[DD]
    - **DEC**: Decrements DD by one.
        - **Symbol**: DEC DD
        - **Logic**: DD = DD - 1
        - **Binary code**: (1011)11[DD]

- **Immediate Load**: Format (11)XXXX[DD].
  - Loads a constant value (immediate) to a register.
  - **LI**: Loads a 4-bit immediate in any GPR.
    - **Symbol**: LI DD, X
    - **Logic**: DD = XXXX
    - **Binary code**: (11)XXXX[DD]

- **Memory access**: Format (100)X[SSDD].
  - Accesses the data memory with input (load) and output (store) operations.
  - **LD**: Loads the value in SS address to the DD register.
    - **Symbol**: LD DD, SS
    - **Logic**: DD = [SS]
    - **Binary code**: (100)0[SSDD]
  - **ST**: Stores SS register value to the address in DD.
    - **Symbol**: ST DD, SS
    - **Logic**: [DD] = SS
    - **Binary code**: (100)1[SSDD]

- **Branching**: Format (10100)X[SS].
    - Branch to an address (of the program memory) in the SS register.
    - The branch is absolute (PC is overwritten by SS), not relative to PC.
    - **B**: Branch to the address in SS register.
        - **Symbol**: B SS
        - **Logic**: PC = SS
        - **Binary code**: (10100)0[DD]
    - **BZ**: If zero flag (ZF) is 1 then branch to the address in SS register.
        - **Symbol**: BZ SS
        - **Logic**: IF ZF IS 1, THEN PC = SS
        - **Binary code**: (10100)1[SS]

- **I/O Access**: Format (10101)X[RR]
    - Controls the I/O port register address in R0.
    - Up to 5 ports: PORT A, PORT B, PORT C, PORT D and PORT E, with 8 pins each (P0 to P7).
    - Even though it explicitly only accepts one operand, it implicitly uses two operands, the other being R0.
    - 3 config registers for every port: DATADIR, DATAOUT and DATAIN.
        - **DATADIR**: 0 for *input* and 1 for *output*.
        - **DATAOUT**: 0 for low output value, and 1 for high.
        - **DATAIN**: 0 for low input value, and 1 for high.
    - **IN**: Read the I/O port value in R0 to DD.
        - **Symbol**: OUT DD
        - **Logic**: DD = [R0]
        - **Binary code**: (10101)0[DD]
    - **OUT**: Write the value in SS to the I/O port in R0.
        - **Symbol**: OUT SS
        - **Logic**: [R0] = SS
        - **Binary code**: (10101)1[SS]

3) Control Signals

- The Program Counter (PC) register operates on the clock’s **falling edge**. This way the PC points to the instruction to be executed in the next **rising edge**. In the falling edge the PC would increment again and point to the next instruction.
- Memory access instruction is performed when MEM_OP is 1.
    - Load operation when LD_ST = 0, DATA_ADDR = SS and DATA_IN = DD.
        - LD R2, R0 ;; where R2 is DD and R0 is SS.
    - Store operation when LD_ST = 1, DATA_ADDR = DD and DATA_OUT = SS.
        - ST R0, R2 ;; where R0 is DD and R2 is SS.

4) Learned Lessons:

- To solve the **branch problem** (the instruction after the branch being executed before the branch’s target instruction) the Program Counter had to be configured to work in the falling edge. The problem is solved because the PC increments in the falling edge and the instruction is then executed in the next rising edge, holding the premise that the PC has to point to the next instruction (albeit not in the current instruction execution, between the rising edge of the execution and the next falling edge of the PC increment, as in the tiny frame PC points to current instruction). This was implemented.
- The use of a falling edge for the increment of the PC and a rising edge for the execution of the instruction proved to be a great match, giving half a cycle of hold of the information in the processing circuit.
- Do not implement the HALT instruction with 0x00 opcode. It is difficult to design the control unit, size the instruction register reset value is interpreted and HALT. It’s better to map NOP instruction to 0x00, and force the CPU to do nothing for one clock cycle. This was not implemented, and *probably* would fix the **branch problem**, where the control unit could replace the instruction after the branch for a NOP in the instruction register, issued via RST signal, but the use of different edges in the PC was a better solution. Still, the exchange of HALT to another opcode and NOP to 0x00 would be a great improvement.

5) Example Codes:

(1) Sum:

```
LI R1, 2        ;; (11)XXXX[DD]      0b11001001   0xC9
LI R2, 3        ;; (11)XXXX[DD]      0b11001110   0xCE
ADD R1, R2      ;; (0)001[SSDD]      0b00011001   0x19
HALT            ;; (00000000)        0b00000000   0x00
```

(2) Sum all and return zero:

```
LI R0, 0        ;; (11)XXXX[DD]      0b11000000   0xC0
LI R1, 1        ;; (11)XXXX[DD]      0b11000101   0xC5
LI R2, 2        ;; (11)XXXX[DD]      0b11001010   0xCA
LI R3, 3        ;; (11)XXXX[DD]      0b11001111   0xCF
ADD R1, R0      ;; (0)001[SSDD]      0b00010001   0x11
ADD R2, R1      ;; (0)001[SSDD]      0b00010110   0x16
ADD R3, R2      ;; (0)001[SSDD]      0b00011011   0x1B
XOR R3, R3      ;; (0)100[SSDD]      0b01001111   0x4F
HALT            ;; (00000000)        0b00000000   0x00
```

(3) Load, sum and store result:

```
LI R0, 0        ;; (11)XXXX[DD]      0b11000000   0xC0
LD R1, R0       ;; (100)0[SSDD]      0b10000001   0x81
LI R0, 1        ;; (11)XXXX[DD]      0b11000100   0xC4
LD R2, R0       ;; (100)0[SSDD]      0b10000010   0x82
ADD R2, R1      ;; (0)001[SSDD]      0b00010110   0x16
LI R0, 2        ;; (11)XXXX[DD]      0b11001000   0xC8
ST R0, R2       ;; (100)1[SSDD]      0b10011000   0x98
HALT            ;; (00000000)        0b00000000   0x00
```

(4) Sum array in memory:

```
0: XOR R0, R0       ;;  (0)101[SSDD]      0b01010000   0x50
1: LD R3, R0        ;; (100)0[SSDD]       0b10000011   0x83
2: XOR R2, R2       ;; (0)101[SSDD]       0b01011010   0x5A
3: INC R0           ;; (1011)10[DD]       0b10111000   0xB8
4: LD R1, R0        ;; (100)0[SSDD]       0b10000001   0x81
5: ADD R2, R1       ;; (0)001[SSDD]       0b00010110   0x16
6: LI R1, B         ;; (11)XXXX[DD]       0b11101101   0xED
7: DEC R3           ;; (1011)11[DD]       0b10111111   0xBF
8: BZ R1            ;; (10100)1[DD]       0b10100101   0xA5
9: LI R1, 3         ;; (11)XXXX[DD]       0b11001101   0xCD
A: B R1             ;; (10100)0[DD]       0b10100001   0xA1
B: XOR R0, R0       ;;  (0)101[SSDD]      0b01010000   0x50
C: LD R3, R0        ;; (100)0[SSDD]       0b10000011   0x83
```

```
D: INC R3          ;; (1011)10[DD]      0b10111011   0xBB
E: ST R3, R2       ;; (100)1[SSDD]      0b10011011   0x9B
F: HALT            ;; (00000000)        0b00000000   0x00
```

(5) Button and LED pins control:

config_gpio:

```
00: XOR R0, R0     ;; (0)101[SSDD]      0b01010000   0x50
01: LI R1, 1       ;; (11)XXXX[DD]      0b11000101   0xC5
02: OUT R1         ;; (10101)1[SS]      0b10101101   0xAD
```

if:

```
03: LI R0, 2       ;; (11)XXXX[DD]      0b11001000   0xC8
04: IN R2          ;; (10101)0[DD]      0b10101010   0xAA
05: AND R2, R0     ;; (0)011[SSDD]      0b00110010   0x32
06: LI R0, E       ;; (11)XXXX[DD]      0b11111000   0xF8
07: BZ R0          ;; (10100)1[DD]      0b10100100   0xA4
```

if:

```
08: LI R0, 1       ;; (11)XXXX[DD]      0b11000100   0xC4
09: IN R2          ;; (10101)0[DD]      0b10101010   0xAA
0A: OR R2, R0      ;; (0)100[SSDD]      0b01000010   0x42
0B: OUT R2         ;; (10101)1[SS]      0b10101110   0xAE
0C: LI R0, 3       ;; (11)XXXX[DD]      0b11001100   0xCC
0D: B R0           ;; (10100)0[DD]      0b10100000   0xA0
```

else:

```
0E: LI R0, 1       ;; (11)XXXX[DD]      0b11000100   0xC4
0F: IN R2          ;; (10101)0[DD]      0b10101010   0xAA
10: NOT R0         ;; (1011)00[DD]      0b10110000   0xD0
11: AND R2, R0     ;; (0)011[SSDD]      0b00110010   0x32
12: LI R0, 1       ;; (11)XXXX[DD]      0b11000100   0xC4
13: OUT R2         ;; (10101)1[SS]      0b10101110   0xAE
14: LI R0, 3       ;; (11)XXXX[DD]      0b11001100   0xCC
15: B R0           ;; (10100)0[DD]      0b10100000   0xA0
```