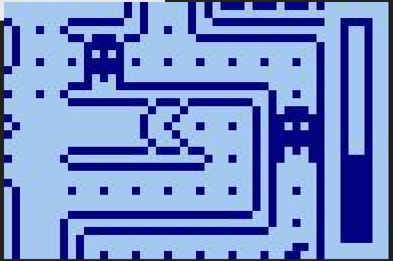
# Sega Dreamcast Visual Memory Unit FPGA Implementation

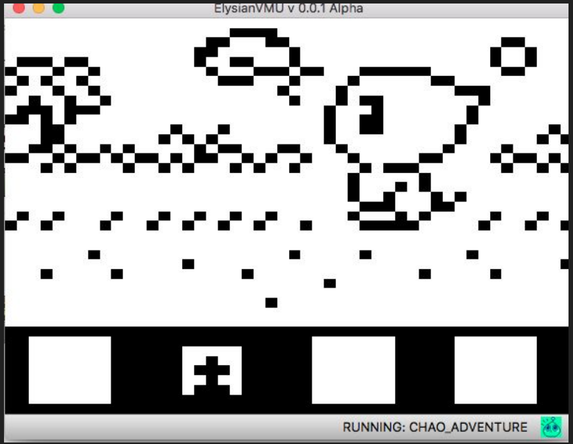
by
Falco Girgis
4/25/17
CPE526

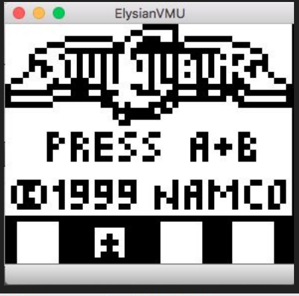
## Visual Memory Unit (VMU)

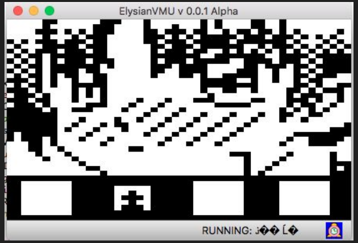


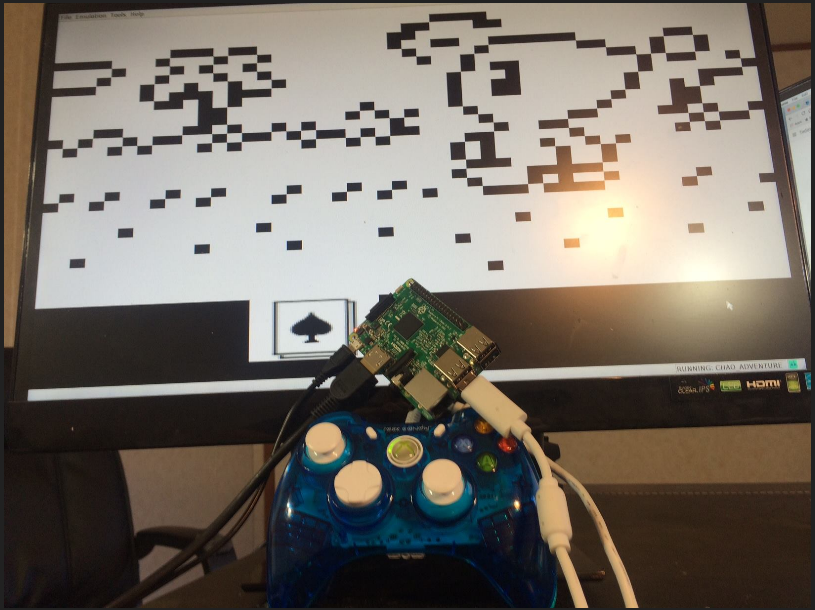
# Cross-Platform VMU Support

- Elysian Shadows Kickstarter (Indie 2D/3D RPG)
  - Windows, Mac, Linux, Sega Dreamcast, Steam, iOS, Android, OUYA, ForgeTV, RaspPi
  - Dreamcast-specific VMU content for display and minigames
- ElysianVMU Software Emulator
  - Play VMU content on any device





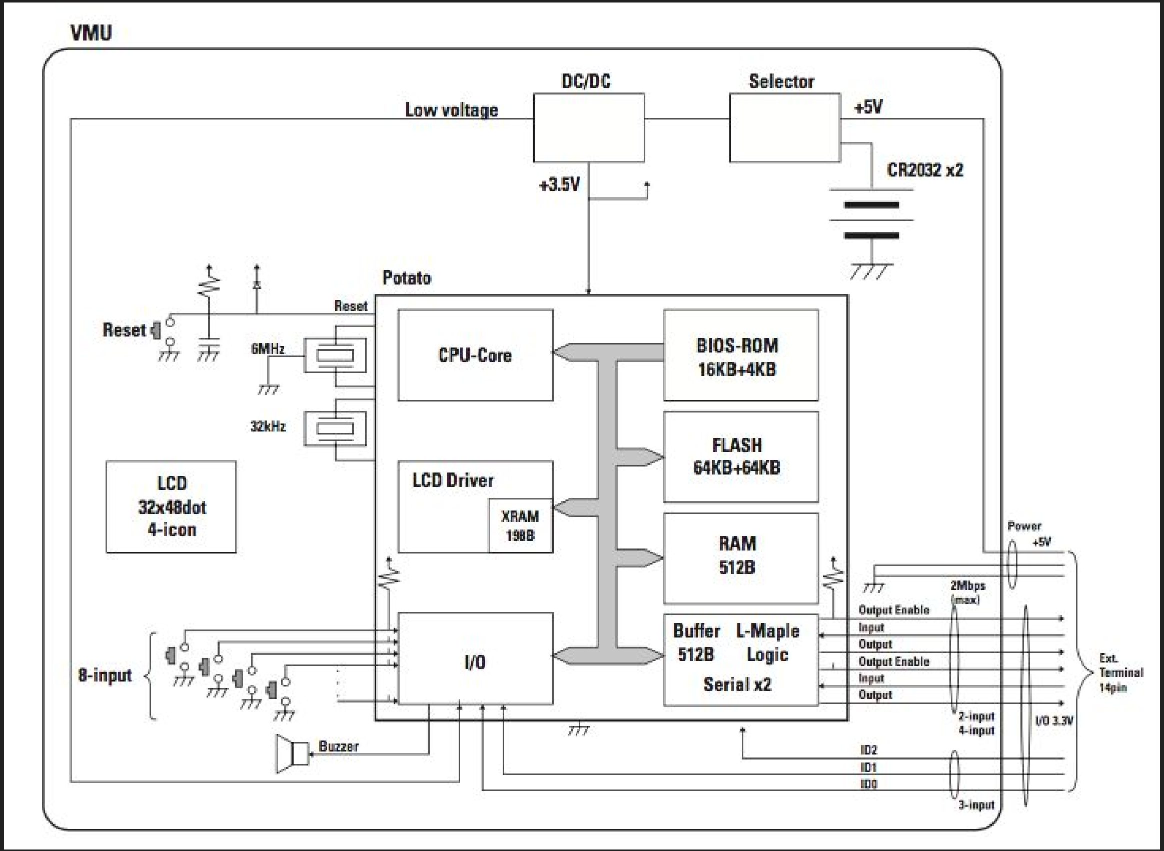




## Next Generation VMU Hardware (VMU2?)

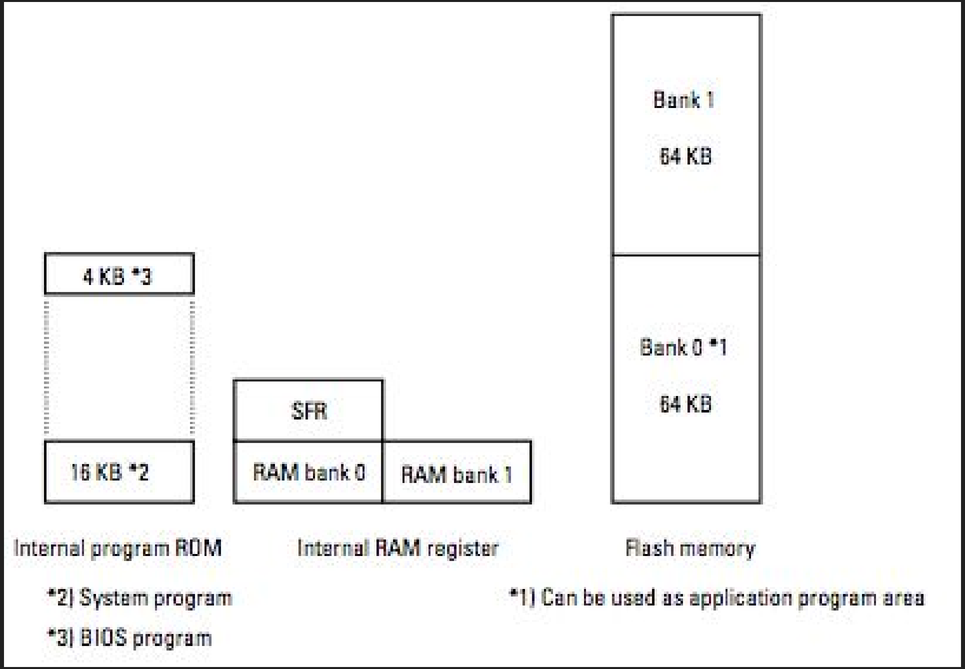
- PC Communication
- USB Storage Device
- Larger Memory Capacity
- Chargeable Batteries
- Backlit LCD
- Higher Resolution
- Color
- High-Quality Audio Output
- 16-bit ISA
- Backwards Compatibility

# System Block Diagram

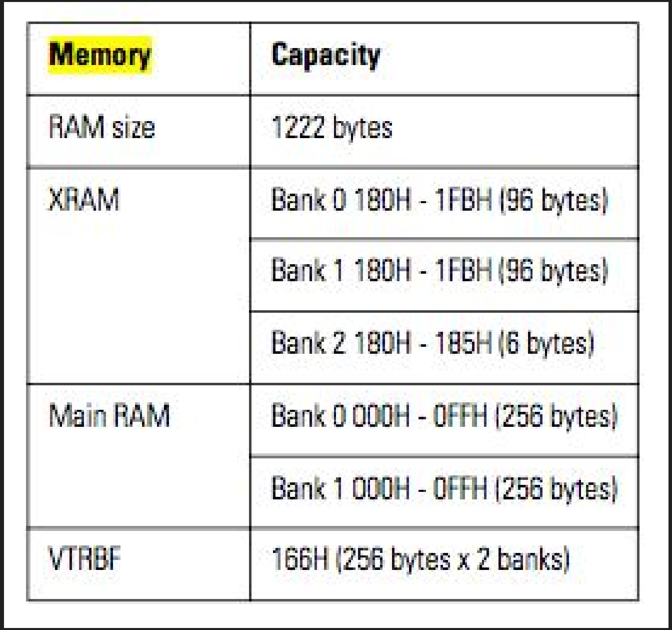


# Memory Layout

## Memory Space

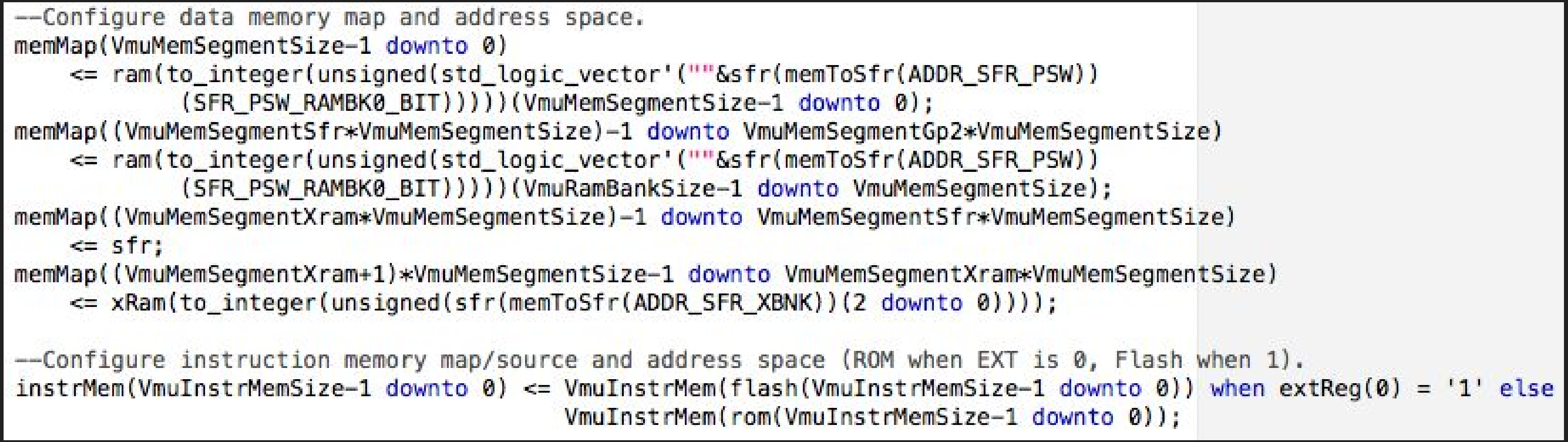


## RAM Space



| Memory | Capacity |
|---|---|
| RAM size | 1222 bytes |
| XRAM | Bank 0 180H - 1FBH (96 bytes) |
| | Bank 1 180H - 1FBH (96 bytes) |
| | Bank 2 180H - 185H (6 bytes) |
| Main RAM | Bank 0 000H - 0FFH (256 bytes) |
| | Bank 1 000H - 0FFH (256 bytes) |
| VTRBF | 166H (256 bytes x 2 banks) |

# Memory Mapping in VHDL



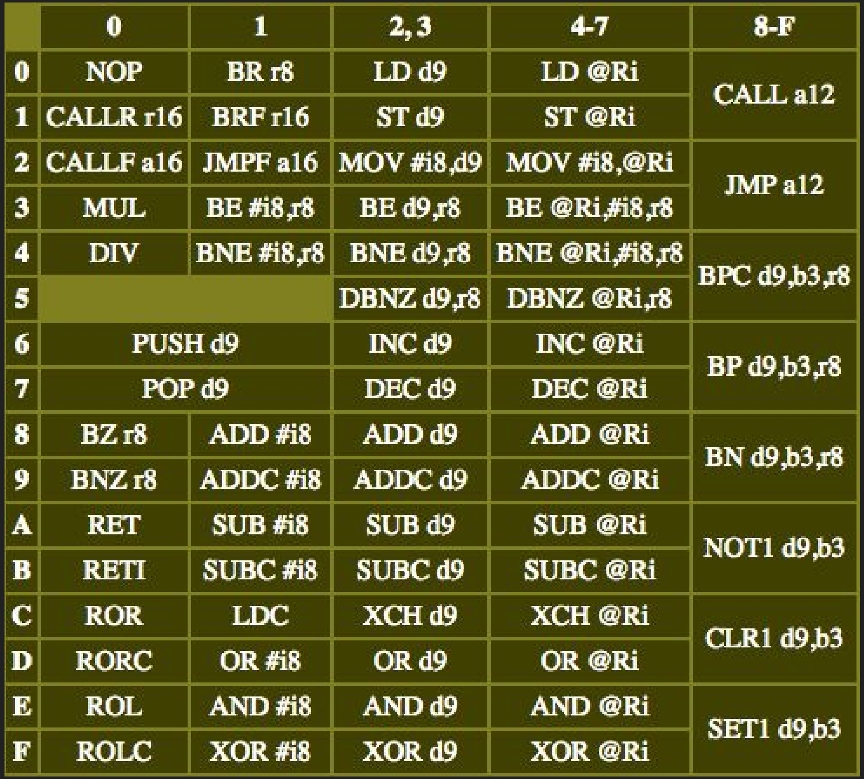
```vhdl
--Configure data memory map and address space.
memMap(VmuMemSegmentSize-1 downto 0)
    <= ram(to_integer(unsigned(std_logic_vector'(""&sfr(memToSfr(ADDR_SFR_PSW))
            (SFR_PSW_RAMBK0_BIT)))))(VmuMemSegmentSize-1 downto 0);
memMap((VmuMemSegmentSfr*VmuMemSegmentSize)-1 downto VmuMemSegmentGp2*VmuMemSegmentSize)
    <= ram(to_integer(unsigned(std_logic_vector'(""&sfr(memToSfr(ADDR_SFR_PSW))
            (SFR_PSW_RAMBK0_BIT)))))(VmuRamBankSize-1 downto VmuMemSegmentSize);
memMap((VmuMemSegmentXram*VmuMemSegmentSize)-1 downto VmuMemSegmentSfr*VmuMemSegmentSize)
    <= sfr;
memMap((VmuMemSegmentXram+1)*VmuMemSegmentSize-1 downto VmuMemSegmentXram*VmuMemSegmentSize)
    <= xRam(to_integer(unsigned(sfr(memToSfr(ADDR_SFR_XBNK))(2 downto 0))));

--Configure instruction memory map/source and address space (ROM when EXT is 0, Flash when 1).
instrMem(VmuInstrMemSize-1 downto 0) <= VmuInstrMem(flash(VmuInstrMemSize-1 downto 0)) when extReg(0) = '1' else
                                        VmuInstrMem(rom(VmuInstrMemSize-1 downto 0));
```

## 8-bit Sanyo LC86K87 CPU Features

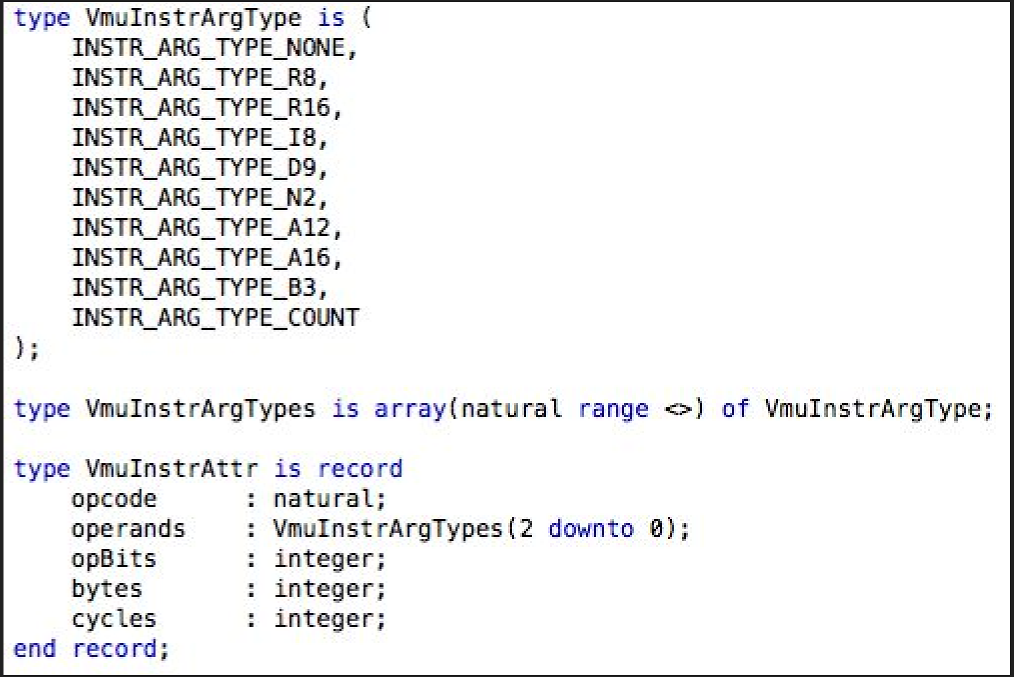
- 128-KB Flash memory
- 20-KB ROM
- 710-byte RAM
- 13-source, 10-vector interrupt architecture
- LCD Controller/Driver
- 16-bit timer/counter/pulse generator
- 16-bit (or 2-channel x 8-bit) synchronous serial interface
- Dedicated Dreamcast interface

# Instruction Map



| | 0 | 1 | 2, 3 | 4-7 | 8-F |
|---|---|---|---|---|---|
| 0 | NOP | BR r8 | LD d9 | LD @Ri | CALL a12 |
| 1 | CALLR r16 | BRF r16 | ST d9 | ST @Ri | |
| 2 | CALLF a16 | JMPF a16 | MOV #i8,d9 | MOV #i8,@Ri | JMP a12 |
| 3 | MUL | BE #i8,r8 | BE d9,r8 | BE @Ri,#i8,r8 | |
| 4 | DIV | BNE #i8,r8 | BNE d9,r8 | BNE @Ri,#i8,r8 | BPC d9,b3,r8 |
| 5 | | | DBNZ d9,r8 | DBNZ @Ri,r8 | |
| 6 | PUSH d9 | | INC d9 | INC @Ri | BP d9,b3,r8 |
| 7 | POP d9 | | DEC d9 | DEC @Ri | |
| 8 | BZ r8 | ADD #i8 | ADD d9 | ADD @Ri | BN d9,b3,r8 |
| 9 | BNZ r8 | ADDC #i8 | ADDC d9 | ADDC @Ri | |
| A | RET | SUB #i8 | SUB d9 | SUB @Ri | NOT1 d9,b3 |
| B | RETI | SUBC #i8 | SUBC d9 | SUBC @Ri | |
| C | ROR | LDC | XCH d9 | XCH @Ri | CLR1 d9,b3 |
| D | RORC | OR #i8 | OR d9 | OR @Ri | |
| E | ROL | AND #i8 | AND d9 | AND @Ri | SET1 d9,b3 |
| F | ROLC | XOR #i8 | XOR d9 | XOR @Ri | |

## VHDL Instruction Attribute Record

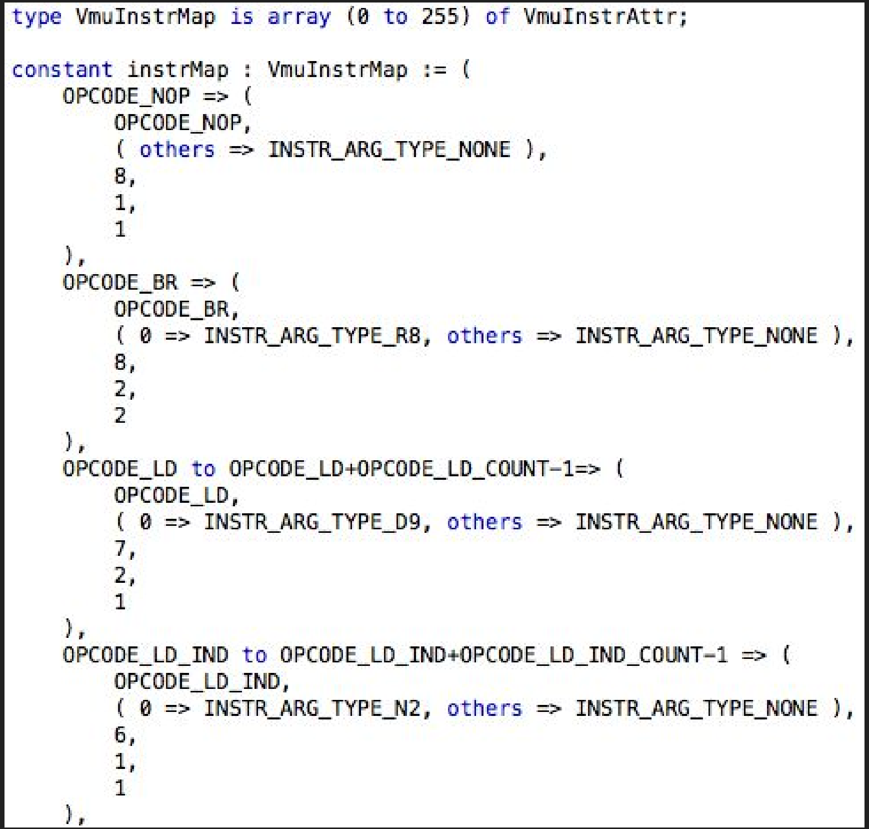


```vhdl
type VmuInstrArgType is (
    INSTR_ARG_TYPE_NONE,
    INSTR_ARG_TYPE_R8,
    INSTR_ARG_TYPE_R16,
    INSTR_ARG_TYPE_I8,
    INSTR_ARG_TYPE_D9,
    INSTR_ARG_TYPE_N2,
    INSTR_ARG_TYPE_A12,
    INSTR_ARG_TYPE_A16,
    INSTR_ARG_TYPE_B3,
    INSTR_ARG_TYPE_COUNT
);

type VmuInstrArgTypes is array(natural range <>) of VmuInstrArgType;

type VmuInstrAttr is record
    opcode      : natural;
    operands    : VmuInstrArgTypes(2 downto 0);
    opBits      : integer;
    bytes       : integer;
    cycles      : integer;
end record;
```

# VHDL Instruction Attribute Lookup Table



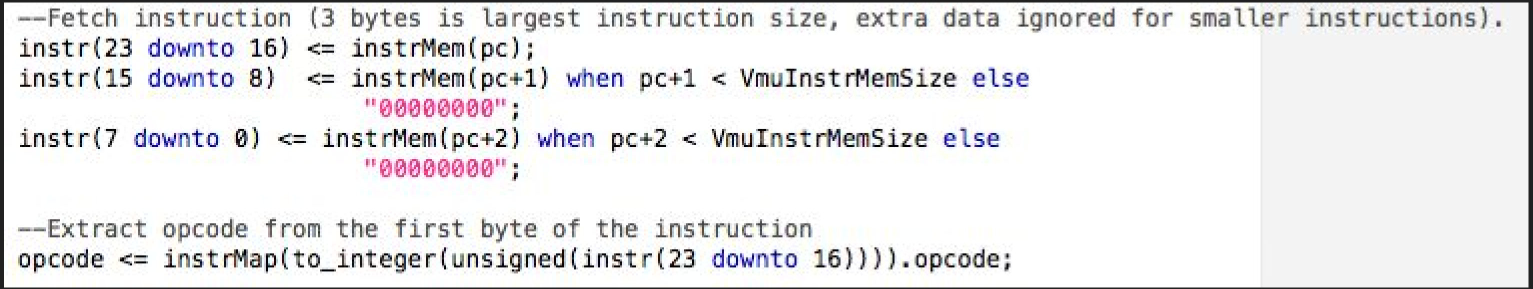
```
type VmuInstrMap is array (0 to 255) of VmuInstrAttr;

constant instrMap : VmuInstrMap := (
    OPCODE_NOP => (
        OPCODE_NOP,
        ( others => INSTR_ARG_TYPE_NONE ),
        8,
        1,
        1
    ),
    OPCODE_BR => (
        OPCODE_BR,
        ( 0 => INSTR_ARG_TYPE_R8, others => INSTR_ARG_TYPE_NONE ),
        8,
        2,
        2
    ),
    OPCODE_LD to OPCODE_LD+OPCODE_LD_COUNT-1=> (
        OPCODE_LD,
        ( 0 => INSTR_ARG_TYPE_D9, others => INSTR_ARG_TYPE_NONE ),
        7,
        2,
        1
    ),
    OPCODE_LD_IND to OPCODE_LD_IND+OPCODE_LD_IND_COUNT-1 => (
        OPCODE_LD_IND,
        ( 0 => INSTR_ARG_TYPE_N2, others => INSTR_ARG_TYPE_NONE ),
        6,
        1,
        1
    ),
```

## VHDL CPU Core Logic

1. Clock Process
   a. Update program counter
   b. Handle reset signal and logic
2. Fetch Instruction
3. Extract Operands
4. Fetch Register Indirect Pointer Operand
5. Execute Instruction

## 1 - Fetch Instruction



```
--Fetch instruction (3 bytes is largest instruction size, extra data ignored for smaller instructions).
instr(23 downto 16) <= instrMem(pc);
instr(15 downto 8)  <= instrMem(pc+1) when pc+1 < VmuInstrMemSize else
                        "00000000";
instr(7 downto 0) <= instrMem(pc+2) when pc+2 < VmuInstrMemSize else
                        "00000000";

--Extract opcode from the first byte of the instruction
opcode <= instrMap(to_integer(unsigned(instr(23 downto 16)))).opcode;
```

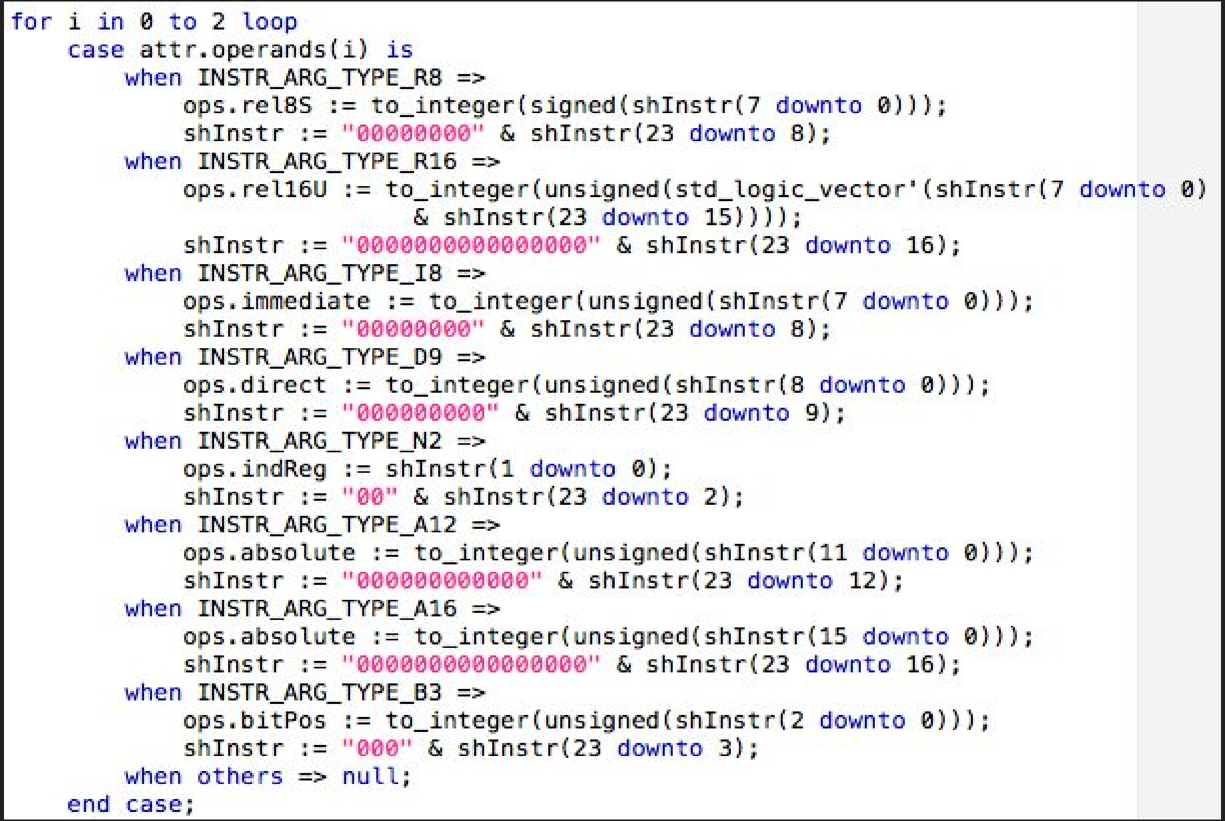
## 2 - Extract Operands

1. Decode instruction for operand types
2. Extract operands from instruction
    a. Types and packing determined by address modes
    b. 2 Different Cases
        i. Specially-coded Instructions
            - Hard-coded logic
        ii. Standard-encoded instructions
            - Standard loop (3 possible operands)

## Addressing Modes

| Mode | Bits | Description |
|---|---|---|
| Immediate | 8 | Operand given in instruction |
| Direct | 9 | Address of operand in memory |
| Indirect | 2 | Index of a special pointer register in memory which contains the address of the operand. |
| Bit specifier | 3 | Specifies a bit offset within a byte operand |
| Absolute | 12/16 | 12 or 16 bits of PC are set to the given address |
| Relative | 8/16 | Address encoded is added to PC |

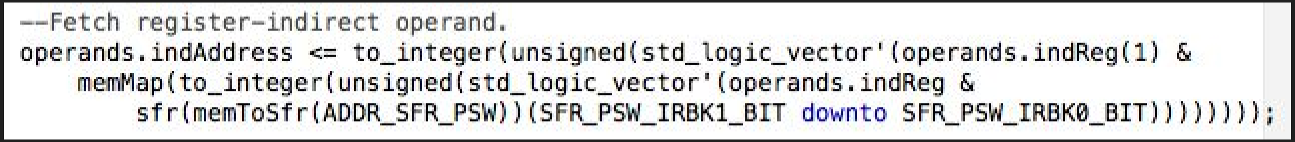
# VHDL General-Case Operand Extraction Logic



```vhdl
for i in 0 to 2 loop
    case attr.operands(i) is
        when INSTR_ARG_TYPE_R8 =>
            ops.rel8S := to_integer(signed(shInstr(7 downto 0)));
            shInstr := "00000000" & shInstr(23 downto 8);
        when INSTR_ARG_TYPE_R16 =>
            ops.rel16U := to_integer(unsigned(std_logic_vector'(shInstr(7 downto 0)
                            & shInstr(23 downto 15))));
            shInstr := "0000000000000000" & shInstr(23 downto 16);
        when INSTR_ARG_TYPE_I8 =>
            ops.immediate := to_integer(unsigned(shInstr(7 downto 0)));
            shInstr := "00000000" & shInstr(23 downto 8);
        when INSTR_ARG_TYPE_D9 =>
            ops.direct := to_integer(unsigned(shInstr(8 downto 0)));
            shInstr := "000000000" & shInstr(23 downto 9);
        when INSTR_ARG_TYPE_N2 =>
            ops.indReg := shInstr(1 downto 0);
            shInstr := "00" & shInstr(23 downto 2);
        when INSTR_ARG_TYPE_A12 =>
            ops.absolute := to_integer(unsigned(shInstr(11 downto 0)));
            shInstr := "000000000000" & shInstr(23 downto 12);
        when INSTR_ARG_TYPE_A16 =>
            ops.absolute := to_integer(unsigned(shInstr(15 downto 0)));
            shInstr := "0000000000000000" & shInstr(23 downto 16);
        when INSTR_ARG_TYPE_B3 =>
            ops.bitPos := to_integer(unsigned(shInstr(2 downto 0)));
            shInstr := "000" & shInstr(23 downto 3);
        when others => null;
    end case;
```

## 3 - Fetch Register Indirect Operand Pointer

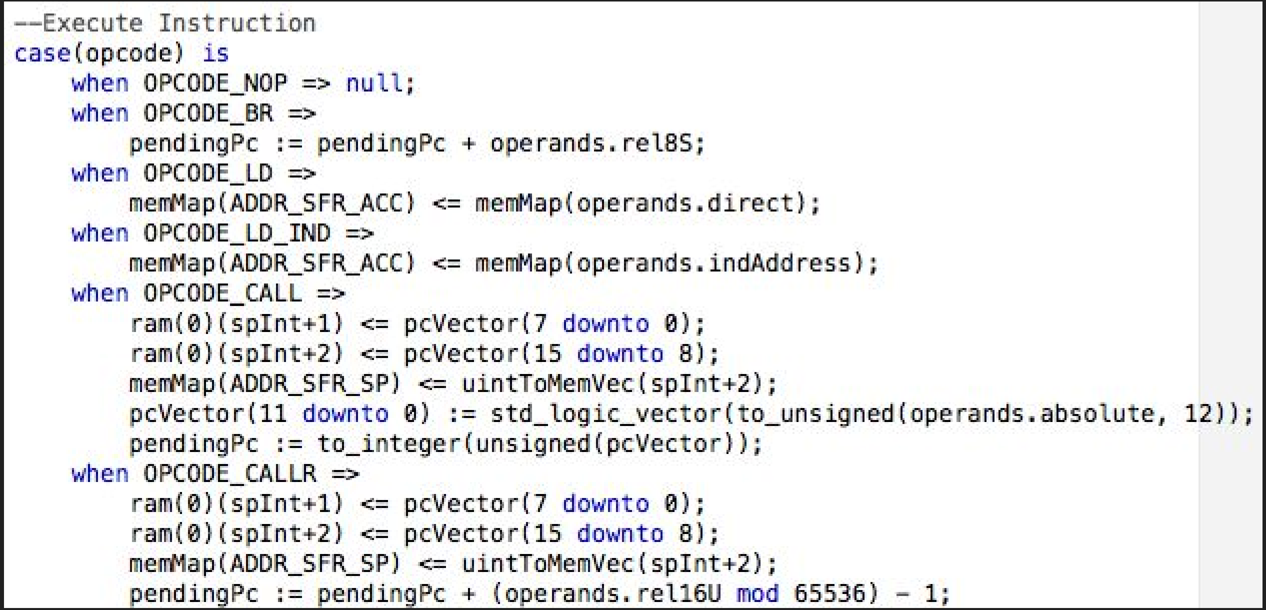
- Not straightforward
  - 8-bit words, 9-bit addresses
  - Pointers stored in special preset addresses
  - Address Calculation
    - Bit 8: MSB of ptr reg index (from instruction)
    - Bit 7 downto 0: Value stored in the given ptr reg



```
--Fetch register-indirect operand.
operands.indAddress <= to_integer(unsigned(std_logic_vector'(operands.indReg(1) &
    memMap(to_integer(unsigned(std_logic_vector'(operands.indReg &
        sfr(memToSfr(ADDR_SFR_PSW))(SFR_PSW_IRBK1_BIT downto SFR_PSW_IRBK0_BIT)))))));
```

## 4 - Execute Instruction

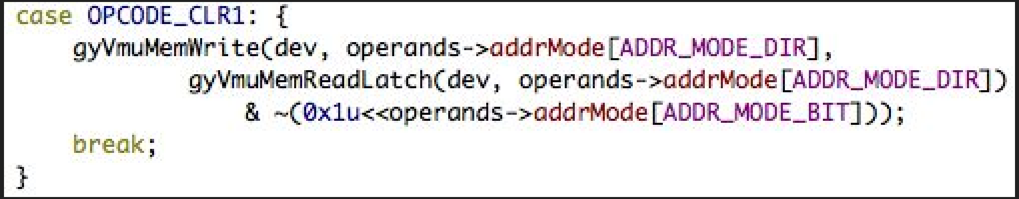
- Weirdly easy



```
--Execute Instruction
case(opcode) is
    when OPCODE_NOP => null;
    when OPCODE_BR =>
        pendingPc := pendingPc + operands.rel8S;
    when OPCODE_LD =>
        memMap(ADDR_SFR_ACC) <= memMap(operands.direct);
    when OPCODE_LD_IND =>
        memMap(ADDR_SFR_ACC) <= memMap(operands.indAddress);
    when OPCODE_CALL =>
        ram(0)(spInt+1) <= pcVector(7 downto 0);
        ram(0)(spInt+2) <= pcVector(15 downto 8);
        memMap(ADDR_SFR_SP) <= uintToMemVec(spInt+2);
        pcVector(11 downto 0) := std_logic_vector(to_unsigned(operands.absolute, 12));
        pendingPc := to_integer(unsigned(pcVector));
    when OPCODE_CALLR =>
        ram(0)(spInt+1) <= pcVector(7 downto 0);
        ram(0)(spInt+2) <= pcVector(15 downto 8);
        memMap(ADDR_SFR_SP) <= uintToMemVec(spInt+2);
        pendingPc := pendingPc + (operands.rel16U mod 65536) - 1;
```

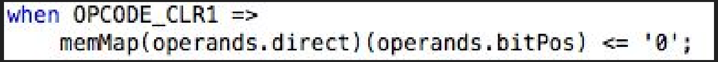
# C vs VHDL Opcode Implementation

C



```
case OPCODE_CLR1: {
    gyVmuMemWrite(dev, operands->addrMode[ADDR_MODE_DIR],
            gyVmuMemReadLatch(dev, operands->addrMode[ADDR_MODE_DIR])
                & ~(0x1u<<operands->addrMode[ADDR_MODE_BIT]));
    break;
}
```

VHDL



```
when OPCODE_CLR1 =>
    memMap(operands.direct)(operands.bitPos) <= '0';
```

## Future Work (Playable)

- Feed display buffer to graphics display
  - XRAM is ready
- Wire controller/input buttons
  - Port 3 logic is implemented and ready
- Work RAM
- Timer0 and Timer1

## Questions?