Computer design

ISA behavioral modeling

P.J. Drongowski
SandSoftwareSound.net

Modelling tips and comments

- Specification versus implementation
  - "What?" versus "how?"
  - Specification must be guideline for implementation
- Styles
  - Specification: Clarity, little physical detail
  - Implementation: Resources, detailed timing
- Host machine artifacts
  - Differences in datatype representation
  - Character data
    - Usually eight bits
    - Inconsistent handling of (un)signed bytes
  - Integer data
    - What is "short" or "long?"
    - Usually two's complement arithmetic
  - Unions
    - Storage may hold `int`, `float`, etc.
    - Should define storage as a union type
    - Make treatment of datatype explicit in expressions
  - Arithmetic and logical operations
    - What is the length of an arithmetic result?
    - Is floating point IEEE standard?
  - Simulation of condition codes
    - No direct access to condition codes from C
    - Example: Carry out from 32-bit addition
      - Can mask and detect carry in short words
      - Must simulate final stage of adder
      - Add low order 31-bits to find carry-in to last stage
      - Add carry-in and high order bits to find carry-out

The SP.1 computer

- A small example
- Simple ISA - 16 instructions
  - Direct addressing only
  - Special purpose registers
  - One condition code flag
  - Three instructions for I/O
- Program stored in read only memory (ROM)
- One input, one output and one exception port

ports	Input<7:0> Output<7:0> InStrobe OutStrobe ReadData<7:0> Address<7:0> Exception<3:0>	Sample stream input Sample stream output Input port data strobe Output port data strobe Memory read data Memory address Exception code output
register	IP<7:0> IR<7:0> AC<7:0> MD<7:0> CR<7:0> SR<7:0>	Instruction pointer Instruction register Accumulator register Intermediate result Counting register Subroutine linkage register
flag	Z	Zero condition code

SP.1 instruction set

NOOP ≡	0	No operation; IP ← IP + 1
GET ≡	1	AC ← Input; IP ← IP + 1
PUT ≡	2	Output ← AC; IP ← IP + 1
ADD ≡	3	AC ← AC + MD; IP ← IP + 1
AND ≡	4	Z ← (AC ∧ M[IP+1]) = 0; IP ← IP + 2
SHL ≡	5	AC ← AC ÷ 2; IP ← IP + 1
SHR ≡	6	AC ← AC × 2; IP ← IP + 1
IMMCR ≡	7	M[IP]<3:0> = 1 ⇒ CR ← M[IP]; Z← 0; IP← IP + 2
IMMAC ≡		M[IP]<3:0> = 2 ⇒ AC ← M[IP]; Z← 0; IP← IP + 2
IMMMD ≡		M[IP]<3:0> = 4 ⇒ MD ← M[IP]; IP← IP + 2
INC ≡	8	CR← CR - 1; Z ← (CR = 0); IP ← IP + 1
SWAP ≡	9	AC ← MD; MD ← AC; IP ← IP + 1
EXC ≡	10	Exception ← IR<3:0>; IP ← IP + 1
BRZ ≡	11	Z = 1 ⇒ IP ← M[IP]
		Z = 0 ⇒ IP ← IP + 1
BR ≡	12	IP ← M[IP];
CALL ≡	13	IP ← M[IP]; SR ← IP
RET ≡	14	IP ← SR
HALT ≡	15	Exception ← IR<3:0>

```
typedef struct rstruct
   {
   union
      {
      SignedByte SB ;
      UnsignedByte UB ;
      }
   Register ;
   }
register ;

typedef struct pstruct
   {
   unsigned int Port : 8 ;
   }
port ;

typedef struct fstruct
   {
   unsigned int Flag : 1 ;
   }
flag ;
```

Architectural simulation

- Header comment to identify author, etc.
- Architectural specification is software and should be treated with respect!

```
/*
 * Signal processing computer simulation
 */

/*
 * Author:  Paul J. Drongowski
 * Address: Computer Engineering and Science
 *          Case Western Reserve University
 *          Cleveland, Ohio 44106
 * CSNet:   pjd@alpha.ces.cwru.edu
 * Version: 1
 * Date:    25 July 1986
 *
 * Copyright (c) 1986 Paul J. Drongowski
 */

#include <stdio.h>
```

Define operation codes

- Define symbolic constants for op-codes
- Use all upper case for constant names
- "`0X`" is the prefix for hexadecimal constants

```
/***************
 * SP op-codes *
 ***************/

#define OP_NOOP 0x0
#define OP_GET  0x1
#define OP_PUT  0x2
#define OP_ADD  0x3
#define OP_AND  0x4
#define OP_SHR  0x5
#define OP_SHL  0x6
#define OP_IMM  0x7
#define OP_DEC  0x8
#define OP_SWAP 0x9
#define OP_EXC  0xA
#define OP_BRZ  0xB
#define OP_BR   0xC
#define OP_CALL 0xD
#define OP_RET  0xE
#define OP_HALT 0xF
```

Define assembly macros

- Define macros to "assemble" SP test program
- Macro parameters for constants and addresses
- Can generate one, two or more instruction values
- Hint: Op-code shifts (<< 4) can be avoided

```
/*******************
 * Assembly macros *
 *******************/

#define NOOP     (OP_NOOP << 4)
#define GET      (OP_GET  << 4)
#define PUT      (OP_PUT  << 4)
#define ADD      (OP_ADD  << 4)
#define AND(V)   (OP_AND  << 4) , V
#define SHR      (OP_SHR  << 4)
#define SHL      (OP_SHL  << 4)
#define IMMCR(V) (OP_IMM  << 4) | 1 , V
#define IMMAC(V) (OP_IMM  << 4) | 2 , V
#define IMMMD(V) (OP_IMM  << 4) | 4 , V
#define DEC      (OP_DEC  << 4)
#define SWAP     (OP_SWAP << 4)
#define EXC(C)   (OP_EXC  << 4) | C
#define BRZ(A)   (OP_BRZ  << 4) , A
#define BR(A)    (OP_BR   << 4) , A
#define CALL(A)  (OP_CALL << 4) , A
#define RET      (OP_RET  << 4)
#define HALT(C)  (OP_HALT << 4) | C
```

Storage declarations

- Display symbolic op-codes during trace
- Declare variable for each architectural register
- One dimensional array to represent general register set
- Beware differing representations of `int`, `char`, etc.
- Primary memory is initialized to test program
  - Declare memory as a one dimensional array
  - Assembly macros expand to instruction values

```
char *Instr[] =

    {
    "Noop","Get", "Put","Add", "And","Shr",
    "Shl", "Imm", "Dec","Swap","Exc","Brz",
    "Br",  "Call","Ret","Halt"
    } ;

char IP,  /* Instruction pointer */
     Z,   /* Zero flag */
     Run, /* Run flag */
     CR,  /* Counting register */
     AC,  /* Accumulator register */
     MD,  /* Temporary register */
     SR;  /* Subroutine return address */

char M[ 256 ] =

    {
    GET,        /* 0   Get test value */
    AND(0xDD),  /* 1,2 Mask test value */
    PUT,        /* Put result */
    HALT(0)     /* Terminate execution */
    };
```

Main procedure

- Display greeting and termination message
- Initialize architectural registers
- Use `Run` flag to control execution loop

```
/***************************
 * Main fetch/execute loop *
 ***************************/

main()

  {
  printf("SP simulator (v1)\n\n");

  Run = 1; /* Set Run flag TRUE */

  IP = 0;  /* Begin at location 0 */
  AC = 0;  /* Clear registers and flag*/
  CR = 0;
  MD = 0;
  Z  = 0;

  while( Run )
    {
    ...
    }

  printf("Leaving the simulator.\n\n");
  }
```

Fetch / execute loop

- Display trace message
- Dispatch to instruction subroutine using op-code
- Display changed processor state

```
while( Run )
  {
  printf("\n%2x %s\n", IP & 0xFF,
         Instr[ (M[iIP] >> 4) & 0xF ]);

  switch( (M[IP] >> 4) & 0xF )
    {
    case OP_NOOP: Noop(); break;
    case OP_GET:  Get();  break;
    case OP_PUT:  Put();  break;
    case OP_ADD:  Add();  break;
    case OP_AND:  And();  break;
    case OP_SHR:  Shr();  break;
    case OP_SHL:  Shl();  break;
    case OP_IMM:  Imm();  break;
    case OP_DEC:  Dec();  break;
    case OP_SWAP: Swap(); break;
    case OP_EXC:  Exc();  break;
    case OP_BRZ:  Brz();  break;
    case OP_BR:   Br();   break;
    case OP_CALL: Call(); break;
    case OP_RET:  Ret();  break;
    case OP_HALT: Halt(); break;
    }

  printf("  AC: %2x CR: %2x\n",
         AC & 0xFF, CR & 0xFF);
  printf("  MD: %2x  Z: %s\n",
         MD & 0xFF, (Z ? "True" : "False"));
  }
```

Instruction procedures

- Show "this state" to "next state" mapping
- Note that each routine changes `IP`
- Simulate communication ports
  - This version uses console input / output
  - Could use files for regression testing

```
Noop()
  {
  IP = IP + 1;
  }

Get()

  {
  static int Value;

  printf("  Input hex value: ");
  scanf("%x", & Value);
  AC = Value;
  IP = IP + 1;
  }

Put()

  {
  printf("  Output hex value: %2x\n",
         (unsigned) AC & 0xFF);
  IP = IP + 1;
  }
```

Instruction procedures

- Show "this state" to "next state" mapping
- Note that each routine changes `IP`
- Clear `Run` flag to stop simulation

```
And()

  {
  if (AC & M[IP + 1]) Z = 0; else Z = 1;
  IP = IP + 2;
  }

Call()

  {
  SR = IP + 2;
  IP = M[IP + 1];
  }

Ret()

  {
  IP = SR;
  }

Halt()

  {
  printf("Halt instruction execution\n");
  Run = 0;
  IP = 0;
  }
```