EE511 Computer Architecture
Project 1: Instruction Set Simulator for Cortex-M0 Processor

Michal Gorywoda

20204787

Spring 2021

1. Introduction

The goal of this project was to create an Instruction Set Simulator (ISS) for a general case 32-bit ARM Cortex-M0 processor. The processor core is compliant with ARMv6-M architecture and uses the most basic 16-bit Thumb instruction set exclusively. A small amount of complex instructions is 32-bit wide.

As mentioned above, the processor is 32-bit wide – that is, its word size, as well as register size is 32 bits or 4 bytes. The core has 16 registers, with R0-12 being general purpose registers, while R13 is a stack pointer (SP), R14 is a link register (LR) and R15 is a program counter (PC). Additionally, there is a program status register (PSR) that contains arithmetic flags – negative (N), zero (Z), carry (C) and overflow (V). Normally, there are more than one program status registers, but in this implementation only one will be used.

A complete ISS is an immensely useful tool for verifying the hardware implementation of processor architecture – it can be used as a reference to compare the results from a physical design to outcome that is expected from instruction set.

In order to verify the ISS, a test program was created as an input to simulator. This allows for manual verification of every instruction behavior. The needs to provide maximum coverage of all instructions and common cases.

Both the simulator and test program will be implemented in C.

2. Simulator

2.1. Design assumptions

The simulator has been designed assuming only privileged thread mode operation – that is, Handler mode is ignored and instructions related to it, as well as any interrupt mechanisms are omitted.

The ISS should cover all of thread mode instructions and simulate their behavior. The instruction set simulator does not provide timing and execution sequence information – otherwise it would have to be related to a specific hardware implementation. The function of ISS is merely to provide results of operations on registers and memory cells.

2.2. Framework

A framework with basic functionalities was provided in advance. The framework included the following files:

- **core.c** – Main file describing execution flow
- **inst.c** – Implementation of auxiliary functions for bit(s) extraction and PC update
- **memory.c** – Memory access functions
- **thumb.c** – Instruction decoding and implementation, as well as function definitions
- **iss.h** – Macros, constants, registers, memory and instruction declarations

2.3. Implementation

The basic functionality of existing framework was as following:

- Initialize registers and memory
- Prompt user for simulation type (run single or multiple instructions, run until breakpoint, view memory, quit simulation)
- Fetch instruction from memory
- Process instruction
- Update PC
- Jump to user prompt

The first step in implementing the instruction behavior was to devise an instruction decoding mechanism. Since the framework for decoding was already present in *void process(uint16_t inst)* function, only its extension was required. This was implemented by extracting bits from specific positions in fetched instruction and comparing them with values defined with macros corresponding to a specific instruction. A nested if-else and switch-case structure was used, as the opcode for Thumb instructions is not constant size. After matching with a macro definition, a call to one and only one function body is performed.

Some of the instructions have unspecified behavior with a specific set of input arguments. Shall these conditions occur, an *assert()* keyword used along with a Boolean expression for these erroneous arguments will break the program execution and display an error message.

The details of instruction related functions are described in comment field above their bodies in *thumb.c* file. In this implementation, a self-commenting code design approach was maintained to minimize the need for additional comments.

The instruction bits encode an information about the functionality and argument sources. Since each instruction type varies in terms of argument bits location, the extraction of these values is done in instruction related functions. The status register is modified at the end of each function that requires flag update.

A type-based instruction classification is described below. Some of the functions are grouped with other type instructions due to similar encoding.

2.3.1. Immediate

Arithmetic-Logic instructions that source one of the arguments from a register and the second argument is contained within instruction as immediate constant. This immediate value can be 3, 5, 7 or 8 bits wide, depending on the instruction.

2.3.2.Register

Arithmetic-Logic instructions whose operands are stored solely in registers. Most of them are 2-operand, that is, Rm is register storing one of the arguments and Rdn is storing second argument and will be written to with result. Some of the instructions have only one argument and others do not store the result.

2.3.3. Load/Store

This type of operation is used to access the memory space. Instruction of this type are able to load data from a specific memory cell to one of the registers, or store register value in a memory cell. A memory address must be provided in order to access the cell under this address – it can be a PC relative with immediate offset (displacement addressing), or stored in another register (register indirect addressing).

Multiple access operations are possible, as well as accessing only lower halfword or lower byte.

2.3.4. Miscellaneous

This subset includes instructions that reverse the order of bytes in register, extend values with zeros or sign as well as push and pop data from stack. Handler mode instructions such as event wait, breakpoint or yield also belong to this class, but are not implemented in this design.

2.3.5. Branch

Instructions that modify the current PC value. They can be performed unconditionally or under specific conditions – in this case, flags in PSR are being checked and jump is performed if condition is true.

In addition, link instructions store the address of the following instruction in LR. It is used for subroutine return, when LR value is written to PC, effectively going back to address following the subroutine call.

An exchange instruction can be executed, which would change the instruction set from Thumb to ARM, or vice-versa. In this application it acts as regular unconditional branch, as only Thumb instruction set is used.

2.3.6. Auxiliary functions

A set of auxiliary functions that are not defined in instruction set are being used to perform addition, logical and arithmetical shifts, bit rotations and counting bits in a given value. For addition, shift and rotate functions, *alu_result_t* structure was created, allowing these functions to return result along with carry and overflow flags simultaneously.

The pseudocode in *ARMv6-M Architecture Reference Manual* was a base for implementing these functions in C. An example of instruction simulation implementation was shown below.

```
void bx(uint16_t inst)
{
  uint32_t m = INST(6,3);
  uint32_t addr;
  assert(m != 15);

  branch = 1;
  addr = R[m] & 0xFFFFFFFE;
  PC = addr;
}
```

```
void strb_reg(uint16_t inst)
{
  uint32_t m = INST(8,6);
  uint32_t n = INST(5,3);
  uint32_t t = INST(2,0);
  uint32_t addr = R[n]+ R[m];

  write_byte(addr, (R[t] & 0x000000FF));
}
```

```
void lsl_reg(uint16_t inst)
{
  uint32_t m = INST(5,3);
  uint32_t n = INST(2,0);
  uint32_t d = n;
  alu_result_t shift_op;

  shift_op = logic_shift_left(R[n], (R[m] &
0x000000FF));

  R[d] = shift_op.result;

  APSR.N = GET_NEGATIVE(shift_op.result);
  APSR.Z = GET_ZERO(shift_op.result);
  APSR.C = shift_op.carry;
}
```

```
void add_imm8(uint16_t inst)
{
  uint32_t n = INST(10,8);
  uint32_t d = n;
  uint32_t imm8 = INST(7,0);
  uint32_t imm32 = zeroExtend32(imm8);
  alu_result_t add_op;

  add_op = add_with_carry(R[n], imm32, 0);
  R[d] = add_op.result;

  APSR.N = GET_NEGATIVE(add_op.result);
  APSR.Z = GET_ZERO(add_op.result);
  APSR.C = add_op.carry;
  APSR.V = add_op.overflow;
}
```

LSL IMM	LDR PCREL	ADR
LSR IMM	STR REG	ADD SPIMM1
ASR IMM	STRH REG	
ADD REG1	STRB REG	ADD SPIMM2
SUB REG	LDRSB REG	SUB SPIMM
ADD 3BIMM	LDR REG	SXTH
SUB 3BIMM	LDRH REG	SXTB
MOV IMM	LDRB REG	UXTH
CMP IMM	LDRSH REG	UXTB
ADD 8BIMM	STR 5BIMM	PUSH
SUB 8BIMM	LDR 5BIMM	CPS
	STRB IMM	REV
AND REG	LDRB IMM	REV16
EOR REG	STRH IMM	REVSH
LSL REG	LDRH IMM	POP
LSR REG	STR 8BIMM	BKPT
ASR REG	LDR 8BIMM	NOP
ADC REG		YIELD
SBC REG	ADD REG2	WFE
ROR REG	CMP REG2	WFI
TST REG	MOV REG1	SEV
RSB REG	BX	
CMP REG1	BLX	STM
CMN REG		LDM
ORR REG	B UNCOND	
MUL REG		B COND
BIC REG	MSR	SVC
MVN REG		
	MRS	BL

Figure 1: List of instructions in design. Yellow labeled instructions not implemented

3. Test program

3.1. Overview

The test program is used to verify the functionality of ISS. It is effectively a C program covering all of the instructions in Thumb set. It is then compiled with ARM GCC and converted into HEX format that can be used as input for simulator that is compiled earlier.

The framework for test program consists of the *test.c* file that includes test cases, assembly file for startup and scripts for compiling and format conversion.

3.2. Instruction coverage

One of main requirements for test program is the maximum coverage of Thumb instructions. As first functional check, all of instructions whose behavior simulation was implemented in ISS were explicitly invoked with all of register value changes recorded.

Secondly, common case subroutines check is desired. For this an implementation of selected frequently used algorithms shall be inserted in test program.

3.3. Implementation

When writing a program in a high-level language, such as C for embedded systems, the programmer has a limited control over specific instructions generated by the compiler. Most of the time, the compiling mechanism will optimize the selection of instructions based on previous commands and arguments for operation. While still programming in C, the designer is able to hint the use of some instructions, or manipulate the optimization settings.

Due to this reason, for a complete Thumb instruction coverage, a *asm volatile()* blocks were used to invoke desired instructions in assembly language. This is not a usual approach to programming embedded systems outside time-critical blocks, but it is the only way to fully test all instruction behavior.

Having that in mind, a second part of test program was created using C functions and cover some of the most known algorithms: Fibonacci sequence computation and Bubble Sort.

Examples of both types of test code are presented in a table below.

```
  asm volatile(
    "LS: mov r2, #0xA5\n\t"
    "mov r0, r2\n\t"
    "lsl r0, #8\n\t"
    "orr r0, r0, r2\n\t"
    "lsl r0, #8\n\t"
    "orr r0, r0, r2\n\t"
    "lsl r0, #8\n\t"
    "orr r0, r0, r2\n\t"
    "mov r1, #2\n\t"
    "lsl r1, #16\n\t"
    "mov r2, #0\n\t"
    "mov r3, #4\n\t"
    "mov r4, #8\n\t"
    "str r0, [r1, r2]\n\t"
    "strh r0, [r1, r3]\n\t"
    "strb r0, [r1, r4]"
  );
```

```
  //ROR
  asm volatile("ror r1, r1, r2");
  //TST
  asm volatile("tst r0, r1");
  //RSB
  asm volatile("neg r1, r1");
  //CMP1
  asm volatile("cmp r3, r1");
```

```
void bubblesort(int array[], int size)
{
  int i, j, temp;

  for(i = 0; i < size-1; i++)
  {
    for(j = 0; j < size-i-1; j++)
    {
      if(array[j] > array[j+i])
      {
        temp = array[j];
        array[j] = array[j+1];
        array[j+1] = temp;
      }
    }
  }
}
```

```
int fibonacci(int n)
{
  int sum, t1, t2, cnt;
  t1 = 0;
  t2 = 1;
  sum = 1;
  for(cnt = 0; cnt < n; cnt++)
  {
    t1 = t2;
    t2 = sum;
    sum = t1 + t2;
  }

  return sum;
}
```

4. Conclusion

After implementation of ISS, the test program was executed in simulated processor and results were compared to expected output list.

All of the implemented instructions were verified with architecture manual and deemed as correct.

The completely verified Instruction Set Simulator will be used as a reference standard for hardware implementation of the processor.