ARM Emulator

DESIGN DOCUMENT

Mayank Thirani | CS 631 | March 12, 2016

Contents

Introduction

ARM emu has been created to emulate the Assembly Language Instructions of Raspberry Pi 2 and additionally recognize the total number of instructions executed, register reads/ writes and performance analysis for each assembly functions.

This design document tutorial guides you through all the functions implemented in ARM Emulator [**armemu.c**] and the working principle of ARM emu. It also lists down all the instructions supported in ARM emu for now. We have tried to derive the working mechanism of ARM emu by emulating the 4 assembly functions – factorial (recursive), factorial (iterative), sum of an array (in recursive way), insertion sort, which were linked to ARM emu dynamically through **gcc**.

Architectural Design of ARM emu

Below flowchart depicts the overall design principle of ARM emu and is invoked for each assembly function that needs to be emulated:

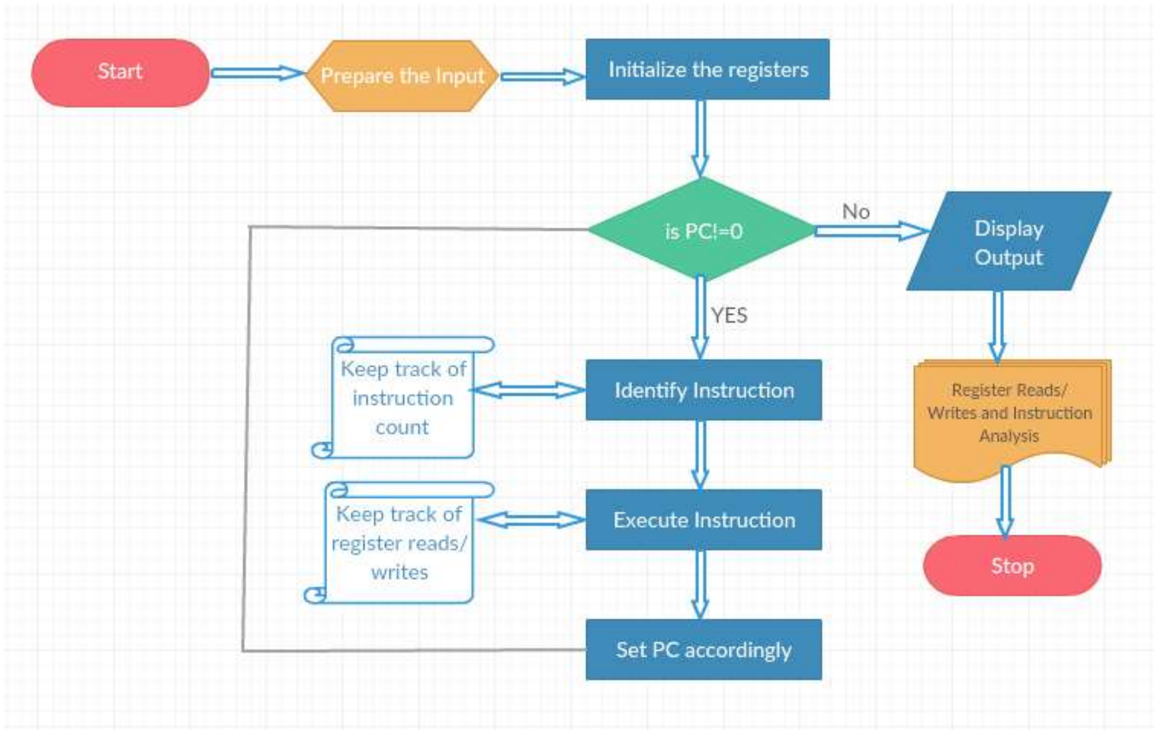


Figure 1 Overall Design Principle of ARM emu

Instructions Supported

Below are the list of Instructions supported in the ARM emu:

- Computation Instructions:
    - MOV
    - ADD
    - SUB
    - CMP
    - MUL
- Memory Instructions:
    - LDR
    - STR
- Branching Instructions:
    - B, BL, BX and B<Cond>
- Shift Operations such as LSL, ASR for all the above except Branching Instructions

Working Principle

BASIC STRUCTURE

```
struct arm_state {
    unsigned regs[16];
    unsigned cpsr;
    unsigned char stack[ARM_STACK_SIZE];
    unsigned regReads[16];
    unsigned regWrites[16];
    unsigned cpsrReads;
    unsigned cpsrWrites;
    unsigned memoryInstr;
    unsigned computeInstr;
    unsigned branchInstr;
};
```

Structure of ARM emu provides the set of 16 Integer registers – **unsigned regs[16]** , a cpsr (current program status register) – **unsigned cpsr**, which were actually used to emulate the 16 Integer registers and cpsr of Raspberry Pi 2 and stack array - **unsigned char stack[ARM_STACK_SIZE]**, is used to emulate the memory referenced by the stack pointer.

Reads/ Writes usage for each register has been stored in the array **regReads [16]**, **regWrites [16]**, **cpsrReads** and **cpsrWrites**.

Different Instructions were computed during the time of execution of ARM assembly functions and were counted in their respective variables – **memoryInstr**, **computeInstr** and **branchInstr**.

INITIALIZATION

To set all the registers and variables to 0, we have used the below function so that it is invoked to reset all the registers and variables before emulating any assembly function.

```
/* Initialize the arm_state struct */
void arm_state_init(struct arm_state *state)
{
    int i;
    for (i = 0; i < 16; i++) {
        state->regs[i] = 0;
        state->regReads[i] = 0;
        state->regWrites[i] = 0;
    }
    state->cpsr = 0;
```

DEBUGGING/ CURRENT ARM STATE

To find out the current status of each registers and value referenced by stack pointer, we can use the below stated function; this is also helpful in debugging purpose.

```
/* Print the arm_state struct */
void arm_state_print(struct arm_state *state)
{
    int i;
    unsigned *ptr;
    for (i = 0; i < 16; i++) {
        printf("regs[%d] = %X\n", i, state->regs[i]);
    }
    ptr = (unsigned *)state->regs[13];
    printf("Stack Value: %X\n", *ptr);
    printf("cpsr = %X\n", state->cpsr);
}
```

BEGIN EMULATION

Ok, now we know the basic structure and initialization of ARM emu. So, we can start emulating each assembly instructions stored in an assembly file [say **rsum.s**] stored somewhere in the path. But first, we have to provide the definition of that function in our ARM emu as shown below at the beginning

```
int rsum(int, int, int, int);
```

Just note as we are passing four arguments in the function of assembly file **rsum.s** so we have used four parameters in our function definition.

Though we have defined the external assembly function, we must be thinking that how to call this external assembly function from ARM emu. Actually calling the function is quite simple, as we can invoke the function inside main() of ARM emu:

```
rv = emu(&state, (void *) rsum, 4, (unsigned *) recurSum);
```

Where recurSum is an array of size 4 as we have to pass 4 arguments in the external assembly function stated earlier and ‘rsum’ argument contains the address of first instruction of the assembly function in rsum.s and ‘state’ is the struct variable of arm_state [structure shown earlier for ARM emu].

What’s next? Now, we are ready to begin our emulation process.

```
/* Function call starts here */
unsigned emu(struct arm_state *state, void *func, int argc, unsigned *args)
{
    int i;

    arm_state_init(state);
```

This function is the starting point for our emulation where it sets the program counter: **state->regs [15]** to the starting address of assembly function pointing to the first instruction, initializes the registers and execute each instruction unless the program counter is set to 0.

```
/* Emulate ARM function */
while(state->regs[15] != 0) {
    emu_instruction(state);
}
```

INSTRUCTION RECOGNIZITION

As in assembly language, different instruction has different set of fields for recognition [Data Processing instruction is recognized by having 0's in bit fields 26 and 27 while Branching instruction is recognized by having 1 in bit field 27, 0 in bit field 26 and 1 in bit field 25], we have to identify correct instruction to emulate them perfectly.

Instruction Recognition starts from the "**emu_instruction**" function which extracts the instruction word pointed to by the program counter in unsigned "**iw**" variable and invokes each supporting function [say **is_b_iw(iw)**] to identify the correct instruction and then call the execution method of identified instruction [say **execute_b_iw(state, iw)**], if the supporting function returns true.

```
/* Determine the correct iw instruction and execute it */
void emu_instruction(struct arm_state *state)
{
    unsigned iw;

    iw = *((unsigned *) state->regs[15]);

    if(is_b_iw(iw)) {
        execute_b_iw(state, iw);
        state->branchInstr = state->branchInstr + 1;
    } else if (is_dt_iw(iw)) {
        execute_dt_iw(state, iw);
        state->memoryInstr = state->memoryInstr + 1;
```

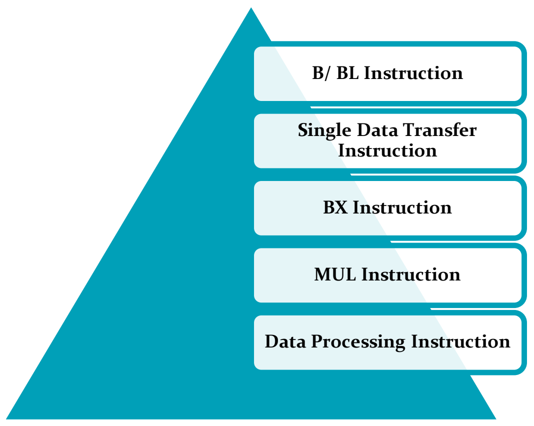
How to identify the instruction(s)?

Let us take an example: Suppose our instruction word "**iw**" contains 11101010000000001111010111011010 then as per the hierarchy, it is first passed to identify it is a B/ BL instruction, which calls the below function. B/ BL instruction is recognized if the bit fields 27, 26, 25 of "**iw**" contains 1, 0 and 1 respectively. Thus we did the RIGHT SHIFT of "**iw**" by 25 bits and performed an AND operation with "**0b101**" where '**ob**' represents that it is binary and returns true/ false accordingly.

```
/* Determine if iw is a branch and link instruction */
bool is_b_iw(unsigned iw)
{
    iw = iw >> 25;
    iw = iw & 0b101;
    return (iw == 0b101);
}
```

Similarly other instructions has been recognized accordingly; let's take another example: Suppose our "**iw**" contains 11100001101000001001000000000010; we can easily identify that it is a data processing instruction by RIGHT SHIFTING the "**iw**" by 26 bits and performing an AND operation with "**0b11**" as we know that if bits 26 and 27 of "**iw**" is 0 then it can be data processing instruction.

Instruction has been recognized based on their hierarchy in ARM instruction set; below pyramid provides the hierarchy where the instruction at bottom is identified at last while instruction at top is identified at first instant.



Instruction being recognized is also counted accordingly. This keeps in track of each different instruction in our assembly function.

INSTRUCTION EXECUTION

Once we have identified the instruction, we have to emulate it accordingly as ARM processor performs. Therefore we call the "execute_***" method [say **execute_b_iw (state, iw)**] accordingly.

In each execute function, we also set the program counter to identify and execute the next instruction in our assembly function; For example in B/ BL instruction, program counter is calculated based on offset value, LDR/ STR/ MOV/ ADD/ SUB/ MUL/ CMP instructions set the program counter to program counter + 4. BX instruction set the program counter to the address contained in link register.

Let’s take an example of one of the execute function say **execute_dp_iw** which corresponds to execute data processing instructions [MOV, ADD, SUB, CMP]

```
/* Execute a data processing instruction word */
void execute_dp_iw(struct arm_state *state, unsigned iw)
{
    unsigned cond;
    unsigned opcode;
    unsigned rn;
    unsigned rm;
    unsigned rd;
    unsigned rotate;
    unsigned imm;
    unsigned immBit;
    unsigned setBit;
    unsigned shiftCode;
    unsigned shiftType;
    unsigned shiftAmount;
    unsigned shiftRegister;
    unsigned valueRmReg;
    int compare;
```

It extracts each fields of “**iw**” by appropriate shifting of bits and based on the value in “**opcode**” variable, it executes that instruction [say **opcode** contains 0b1101] then it identifies that it is MOV instruction and performs the functionality of MOV instruction by copying the value stored in operand2 to destination register.

```
if (opcode == 0b1101) {     //MOV Instruction
    state->regs[rd] = valueRmReg;
    state->regWrites[rd] = state->regWrites[rd] + 1;
} else if (opcode == 0b0100) {      //ADD Instruction
    state->regs[rd] = state->regs[rn] + valueRmReg;
    state->regReads[rn] = state->regReads[rn] + 1;
    state->regWrites[rd] = state->regWrites[rd] + 1;
} else if (opcode == 0b0010) {      //SUB Instruction
    state->regs[rd] = state->regs[rn] - valueRmReg;
    state->regReads[rn] = state->regReads[rn] + 1;
    state->regWrites[rd] = state->regWrites[rd] + 1;
} else if (opcode == 0b1010) {      //CMP Instruction
    compare = state->regs[rn] - valueRmReg;
    state->regReads[rn] = state->regReads[rn] + 1;
```

valueRmReg variable contains the immediate value or the value stored in Rm register, if I bit is set to 0 in "**iw**".

If the operand2 is a register then it also identifies if the shifting operation should be applied to Rm register. If yes then it calculates the valueRmReg accordingly. Two shift operation is supported: **LSL** and **ASR** which is identified by 00 and 10 respectively.

```
if(immBit == 0) {                   //Operand2 is a register
    valueRmReg = state->regs[rm];
    state->regReads[rm] = state->regReads[rm] + 1;
    shiftCode = (iw >> 4) & 0b1;
    shiftType = (iw >> 5) & 0b11;
    if(shiftCode == 1) {
            shiftAmount = (iw >> 8) & 0b1111;
            shiftAmount = state->regs[shiftAmount];
            state->regReads[shiftAmount] =  state->regReads[shiftAmount] + 1;
    }
    else {
            shiftAmount = (iw >> 7) & 0b11111;
    }
    if((shiftType == 0b00) && (shiftAmount != 0)) {
             valueRmReg = valueRmReg * shiftAmount * 2;
    } else if((shiftType == 0b10) && (shiftAmount != 0)) {
             valueRmReg = valueRmReg / (shiftAmount * 2);
    }
}
```

In each execute instruction, we are also keeping track of the reads/ writes operation performed on the register by computing **regReads** and **regWrites**.

Also program counter is set to program counter + 4 in this execute function.

Let's take another example of execute function: say **execute_b_iw** which corresponds to execute data processing instructions [B/ BL]

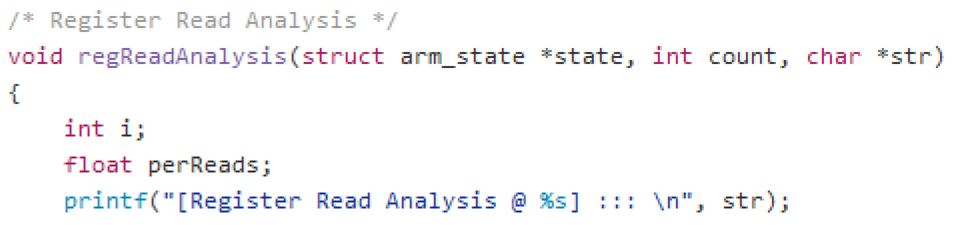
```
/* Execute a branch and link instruction word */
void execute_b_iw(struct arm_state *state, unsigned iw)
{
    unsigned cond;
    unsigned link;
    unsigned offset;
    unsigned offsetCheck;
    int newOffset;

    cond = iw >> 28;
    link = (iw >> 24) & 0b1;
    offset = iw & 0xFFFFFF;
    offsetCheck = (offset >> 23) & 0b1;
```

Offset, Link, Condition fields are extracted here; Offset value, last 24 bits of "**iw**" has been extracted and is used to set the program counter to that address. Link bit, 24th bit of "**iw**" has been extracted to identify if there is a function call or not. Condition fields are used when we have branching instruction with conditions say **BNE, BEQ, BGT, BLT.**

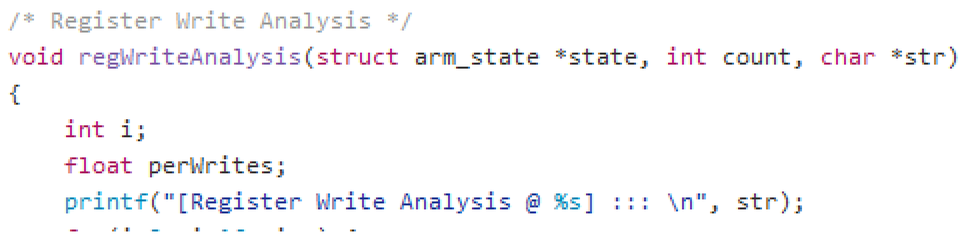
Based on the Branching instruction identified, program counter has been set accordingly. Also note that we are keeping track of reads/ writes of each register.

RESULT ANALYSIS

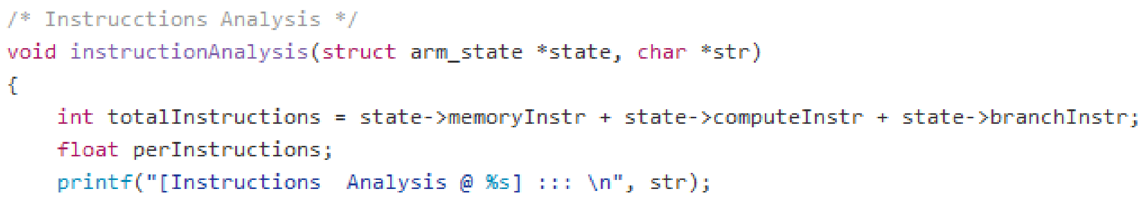
Once we are done with execution of all the instructions of each assembly function, we are basically calling analysis functions to analyze each register read/ write usage, different instruction counts and computing the total instructions executed in each assembly function.



```
/* Register Read Analysis */
void regReadAnalysis(struct arm_state *state, int count, char *str)
{
    int i;
    float perReads;
    printf("[Register Read Analysis @ %s] ::: \n", str);
```

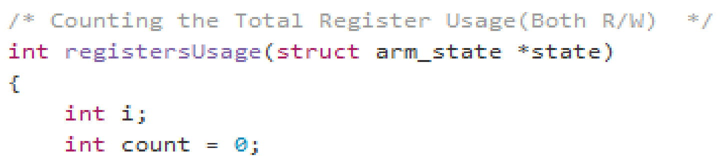


```
/* Register Write Analysis */
void regWriteAnalysis(struct arm_state *state, int count, char *str)
{
    int i;
    float perWrites;
    printf("[Register Write Analysis @ %s] ::: \n", str);
```



```
/* Instrucctions Analysis */
void instructionAnalysis(struct arm_state *state, char *str)
{
    int totalInstructions = state->memoryInstr + state->computeInstr + state->branchInstr;
    float perInstructions;
    printf("[Instructions  Analysis @ %s] ::: \n", str);
```

NOTE: **count** variable determines the total register usage [both reads/ writes] by invoking the below function:



```
/* Counting the Total Register Usage(Both R/W)  */
int registersUsage(struct arm_state *state)
{
    int i;
    int count = 0;
```

Assembly Linking

Assembly functions in ARM emu are linked dynamically while compiling the armemu.c through gcc. First we have to create the <assembly function>.o (object file) by calling as –o <assembly function>.o <assembly function>.s.

Let's consider an example of **rsum.s.** How to create the .o (object) file?

- **as –o rsum.o rsum.s**

Now link the object file to armemu.c. How to do that?

- **gcc –o armemu armemu.c rsum.o**

Appendix

Some key terms used in the document:

- **iw** = instruction word
- **PC** = program counter which is state->regs[15]
- **LR** = link register which is state->regs[14]
- **SP** = stack pointer which is state->regs[13]
- **ARM emu** = ARM emulator