ENGR-E 399/599: Embedded Systems Reverse Engineering

Lecture 1.5: ARM assembly

Austin Roach
ahroach@iu.edu

January 13, 2022

Today's lecture

- ARM assembly basics
- Demo programs
- Bonus topic: Thumb-mode

ARM history

- Advanced/Acorn RISC Machine (1985)
- Various improvements and changes to ISA over the years
- Arm Holdings develops architecture, licenses to other companies
- Most widely deployed ISA
  - Averaged 22 billion Arm chips/year from 2017-2020

Why we care

- Very common instruction set architecture in embedded systems
  - Cortex-M (microcontroller)
  - Cortex-R (real-time processing)
  - Cortex-A (application processor)
- We will focus on 32-bit Arm ISA
  - 64-bit increasingly common in general-purpose computing/mobile applications

ARM/Thumb modes

- Standard ARM mode has fixed-width 32-bit instructions
- Thumb mode was introduced in 1994 with fixed-width 16-bit instructions
  - The masses complained: Too many instructions are missing!
- Thumb-2 mode introduced in 2003 with mixed 16- and 32-bit instructions
  - More instructions, but now variable width instruction set
- Most processors can switch between modes
  - Current processor state indicated by a bit in the status register
- Most instruction mnemonics the same
- ARM mode for this lecture and assignment

General purpose registers

Sixteen 32-bit general purpose registers available at one time:

- r0, r1, r2, r3, … , r15

Alternate symbols:

Register number	Alternate symbol	Description
r9	SB	static base
r10	SL	stack limit
r11	FP	frame pointer
r12	IP	intra-procedure scratch register
r13	SP	stack pointer
r14	LR	link register
r15	PC	program counter

Signed and unsigned integers

Unsigned integers can have values from 0 to 0xffffffff (0 to 4,294,967,295)

Signed numbers represented using two's complement:

`2,147,483,647`	`0x7fffffff`
`10,000,000`	`0x00989680`
`10`	`0x0000000a`
`1`	`0x00000001`
`0`	`0x00000000`
`-1`	`0xffffffff`
`-10`	`0xfffffff6`
`-10,000,000`	`0xff676980`
`-2,147,483,648`	`0x80000000`

Current program status register

Relevant flags:

- `N`: Negative condition flag (bit 31 of the result of instruction)
- `Z`: Zero condition flag (result of instruction was zero)
- `C`: Carry condition flag (instruction results in a carry)
- `V`: Overflow condition flag (instruction results in a signed overflow)

Other registers

- Coprocessor registers
- Additional status registers
- Configuration registers
- Registers for floating-point units
- Registers for SIMD units
- etc.

Data movement: registers

`mov r0, r2`	Register-to-register (copy contents of `r2` to `r0`)
`cpy r0, r2`	Synonym for register-to-register move
`mov r1, #0`	Immediate-to-register (set the contents of `r1` to 0)
`movs r0, r2`	Register-to-register, updating condition flags
`moveq r0, #3`	Conditional move immediate to register, if zero-flag is set

Data movement: load memory

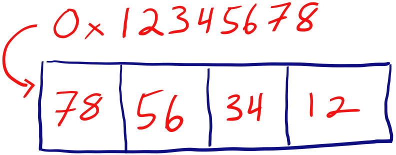
`ldr r0, [r2]`	Load 4 bytes from address in `r2` to `r0`
`ldr r0, [r2, #4]`	Load 4 bytes from address `r2 + 0x4` into `r0`
`ldrb r0, [r2]`	Load 1 byte from address `r2` to `r0`
`ldrb r0, [r2], #1`	Load 1 byte from address in `r2` to `r0`; Update `r2 = r2 + 1` (post-indexed addressing)
`ldrb r0, [r2, #1]!`	Load 1 byte from address `r2 + 1` to `r0`; `r2 = r2 + 1` (pre-indexed addressing)

Data movement: store memory

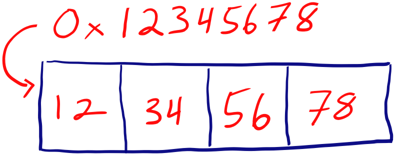
`str r0, [r2]`	Store 4 bytes from `r0` to address in `r2`
`str r0, [r2, #4]`	Store 4 bytes from `r0` to address `r2 + 0x4`
`strb r0, [r2]`	Store 1 byte from `r0` to address `r2`
`strb r0, [r2], #1`	Store 1 byte from `r0` to address in `r2`; Update `r2 = r2 + 1` (post-indexed addressing)
`strb r0, [r2, #1]!`	Store 1 byte from `r0` to address `r2 + 1`; `r2 = r2 + 1` (pre-indexed addressing)

Data storage in memory

Little endian:



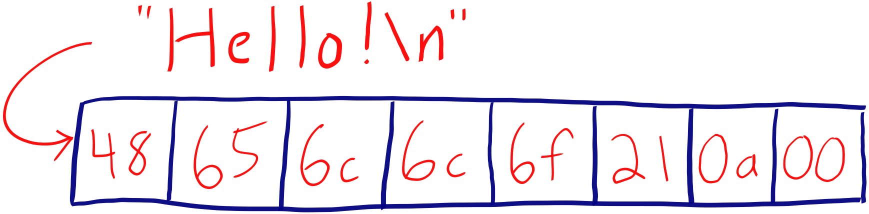
Big endian:



Our code will be for a little endian ARM processor

C string storage in memory

- ASCII characters stored as single-byte (man ascii)
- Strings are null-terminated



"Hello!\n"

48	65	6c	6c	6f	21	0a	00

Arithmetic

`add r1, r1, #0x40`	Add 0x40 to the value in `r1`, storing result in `r1`
`adds r3, r1, r2`	Add `r1` to `r2`, storing result in `r3` and updating the `cpsr` flags
`addeq r1, r1, #0x40`	If the zero flag is set, add 0x40 to the value in `r1`
`sub r1, r1, #0x40`	Subtract 0x40 from `r1`, storing the result in `r1`
`lsl r2, r2, #2`	Logical shift left value in `r2` by 2, storing result in `r2`

Barrel shifter

ARM cores incorporate a barrel shifter to optionally shift register contents:

`add r0, r1, r2, lsr #4`	`r0 = r1 + r2 >> 4`
`ldr r0, [r4, r2, lsl #2]`	Load `r0` with memory contents of `r4 + r2 << 2`

Control flow statements (setting flags)

`cmp r1, r0` Subtract `r0` from `r1`. Discard the result, but update flags.

`cmp r1, #4` Subtract 4 from `r1`. Discard the result, but update flags.

`movs r1, r2` Copy contents of `r2` to `r1`. Update flags based on value.

Condition codes

`EQ`	Equal	Z==1
`NE`	Not equal	Z==0
`HI`	Unsigned higher	C==1 and Z==0
`LS`	Unsigned lower or same	C==0 or Z==1
`GE`	Signed greater than or equal	N==V
`GT`	Signed greater than	Z==0 and N==V
`LE`	Signed less than or equal	Z==1 or N!=V
`LT`	Signed less than	N!=V

Others as defined in the reference manual

Branches

`b 0x1110`	Branch unconditionally to 0x1110
`beq 0x1110`	Branch to 0x1110 is zero flag is set
`bx r0`	Branch to address and exchange (change to Thumb mode if lowest bit of register set, ARM if not set)
`addls pc, pc, r3, lsl 2`	If unsigned lower or same, add `r3 * 4` to PC Note: In ARM mode the PC value is the address of the current instruction + 8

Stack

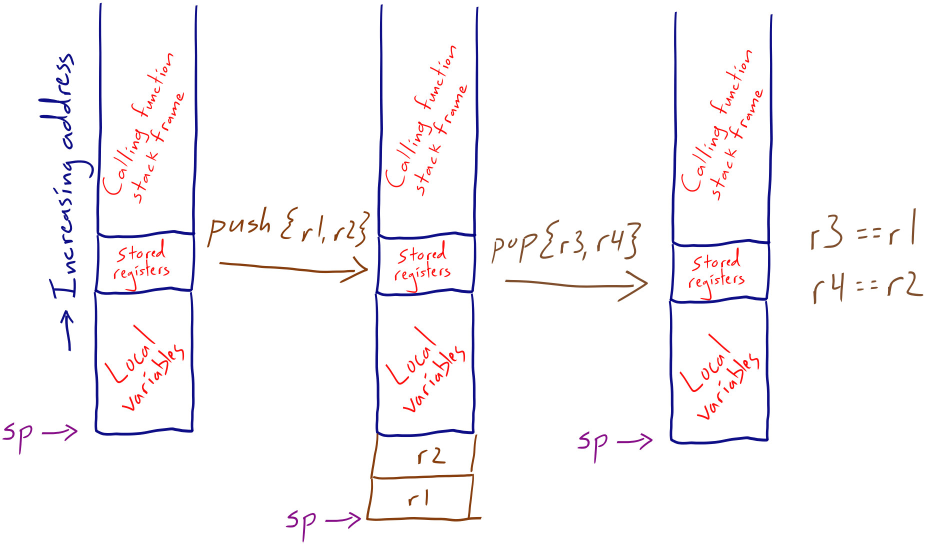
```
push {r4, r11, lr}
stmdb sp!, {r4, r11, lr}
```

Push `lr`, then `r11`, then `r4` onto the stack

```
pop {r4, r11, pc}
ldmia sp!, {r4, r11, pc}
```

Pop first value from stack into `r4`, next value into `r11` then next value into `pc`

Stack frames



Function calls

`bl 0x1110` Branch to target and store the return address in the link register

`blx 0x1110` Branch to target, store the return address in the link register, and change processor mode (ARM/Thumb)

`blx r0` Branch to address in `r0`, store the return address in the link register, and set processor mode to ARM if `r0[0] == 0`, Thumb if `r0[0] == 1`

`bx lr` Branch to address in `lr`, and set processor mode to ARM if `r0[0] == 0`, Thumb if `r0[0] == 1`

Also common to return by popping stored `lr` value directly into `pc`

Calling convention (simplified)

- Integer/pointer arguments passed in `r0`, `r1`, `r2`, and `r3`. Additional arguments passed on stack.
- Integer/pointer values returned in `r0`.

Program examples

ARM Thumb mode

- Previously discussed instructions in 32-bit ARM mode
  - Instructions encoded with a fixed width of 32 bits
- Some processors can use the ARM and Thumb-1/2 instructions
- Some processors operate *only* in Thumb-1/2 mode
- Many of the mnemonics are the same, but encodings are more variable in Thumb mode
- Adds `TBB` (Table Branch Byte) and `TBH` (Table Branch Halfword) instructions for jump tables – don't think you will encounter these
- Adds `CBNZ` (Compare and Branch on Nonzero) and `CBZ` (Compare and Branch on Zero) instructions
- Differences relevant to reverse-engineering mainly involve conditional execution
  - Many instructions always set status flags
    - Proper assembly would still include the `S` suffix, but Ghidra often elides these
  - Very small set of instructions include conditional execution field
    - Special `IT` instruction implements "if-then-else" blocks for conditional execution of other instructions

32-bit ARM conditional execution: Review

Instructions with `S` flag update status flags:

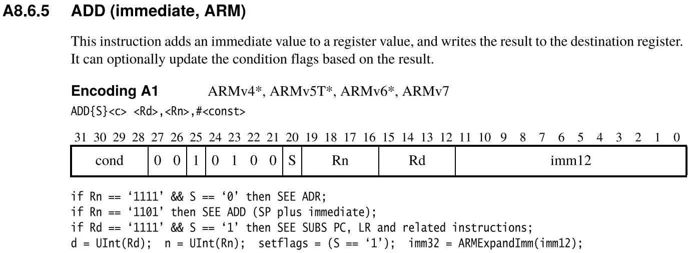
- `add r1, r1, #0x40` – Don't update status flags
- `adds r1, r1, #0x40` – Update status flags

(Almost) any instruction can be conditionaly executed:

- `b 0x1110` – Unconditional branch
- `beq 0x1110` – Conditional branch
- `addne r1, r1, #0x40` – Conditional add
- `movls r2, r1` – Conditional move

ADD (immediate) – ARM encoding

- Only a single encoding
- Conditional execution field for each instruction
- Optionally update status flags for each instruction



A8.6.5 ADD (immediate, ARM)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

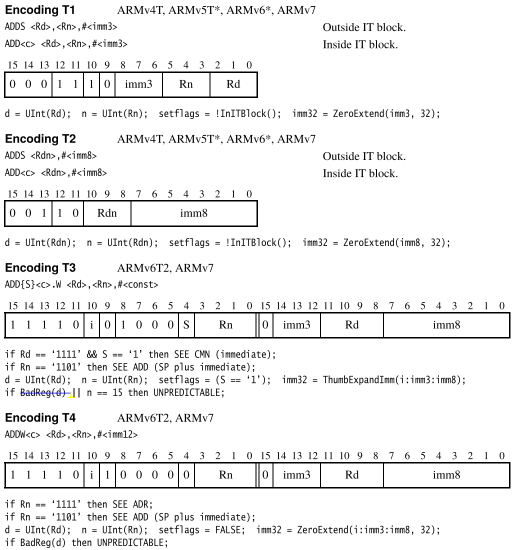
`ADD{S}<c> <Rd>,<Rn>,#<const>`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	1	0	1	0	0	S	Rn				Rd				imm12											

```
if Rn == '1111' && S == '0' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = ARMExpandImm(imm12);
```

ADD (immediate) – Thumb encoding

- Multiple instruction encodings
- Status flags always updated outside IT block in 16-bit encodings
- Status flags optionally updated in only one 32-bit encoding
- Conditional execution imposed by IT block – not encoded in instruction



Encoding T1 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADDS <Rd>,<Rn>,#<imm3>                    Outside IT block.
ADD<c> <Rd>,<Rn>,#<imm3>                  Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 ARMv4T, ARMv5T*, ARMv6*, ARMv7

```
ADDS <Rdn>,#<imm8>                        Outside IT block.
ADD<c> <Rdn>,#<imm8>                      Inside IT block.
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Encoding T3 ARMv6T2, ARMv7

```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	1	0	0	0	S	Rn				0	imm3			Rd				imm8							

```
if Rd == ‘1111’ && S == ‘1’ then SEE CMN (immediate);
if Rn == ‘1101’ then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == ‘1’);  imm32 = ThumbExpandImm(i:imm3:imm8);
if ~~BadReg(d)~~ || n == 15 then UNPREDICTABLE;
```

Encoding T4 ARMv6T2, ARMv7

```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	0	0	0	Rn				0	imm3			Rd				imm8							

```
if Rn == ‘1111’ then SEE ADR;
if Rn == ‘1101’ then SEE ADD (SP plus immediate);
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if BadReg(d) then UNPREDICTABLE;
```

IT instructions

IT

If Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

Example:

```
CMP R0,#0x10
ITETE EQ
ADDEQ R1,R0      <-- Executes if zero flag set
SUBNE R2,R0      <-- Executes if zero flag not set
MOVEQ R3,R1      <-- Executes if zero flag set
MOVNE R3,R2      <-- Executes if zero flag not set
```

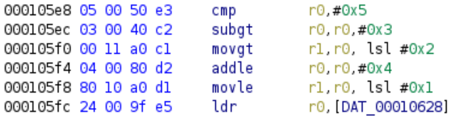
Example application

```
int main(int argc, char **argv)
{
        int input;
...
        input = atoi(argv[1]);

        if (input > 5) {
                input -= 3;
                input *= 4;
        } else {
                input += 4;
                input *= 2;
        }
...
}
```

Example application: ARM vs Thumb-2 conditional execution

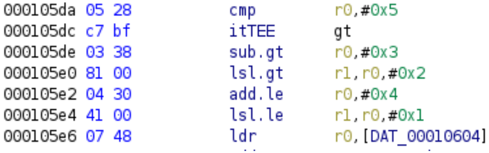
ARM mode



```
000105e8 05 00 50 e3     cmp        r0,#0x5
000105ec 03 00 40 c2     subgt      r0,r0,#0x3
000105f0 00 11 a0 c1     movgt      r1,r0, lsl #0x2
000105f4 04 00 80 d2     addle      r0,r0,#0x4
000105f8 80 10 a0 d1     movle      r1,r0, lsl #0x1
000105fc 24 00 9f e5     ldr        r0,[DAT_00010628]
```

- Conditional execution bits for each instruction

Thumb-2 mode



```
000105da 05 28          cmp        r0,#0x5
000105dc c7 bf          itTEE      gt
000105de 03 38          sub.gt     r0,#0x3
000105e0 81 00          lsl.gt     r1,r0,#0x2
000105e2 04 30          add.le     r0,#0x4
000105e4 41 00          lsl.le     r1,r0,#0x1
000105e6 07 48          ldr        r0,[DAT_00010604]
```

- Conditional execution condition set by `IT` instruction
- Conditional execution mnemonic suffixes provided for ease of reading

If you only pay attention to one Thumb-mode slide...

In mixed-mode environments, disassemblers will sometimes use the wrong mode to dissasemble code!

Results: complete garbage

You may need to correct this manually. In Ghidra, F11 will disassemble in ARM mode, and F12 will disassemble in Thumb mode