ARM

Assembly Language and Machine Code

Goal: Blink an LED
Summary

You need to understand how processors represent and execute instructions.

Instruction set architecture often easier to understand by looking at the bits. Encoding instructions in 32-bits requires trade-offs, careful design.

Only write assembly when it is needed. Reading assembly more important than writing assembly. Allows you to see what the compiler and processor are actually doing.

Normally write code in C (Starting next lecture)
\textbf{ALU only operates on registers}

\textbf{Registers are also 32-bit words}

\textbf{Example:}

\texttt{add r0, r1, r2}

\[ r0 = r1 + r2 \]
add r0, r1, #1

Immediate Value (#1) stored in INST
Load and Store Instructions
Load from Memory to Register (LDR)

\[ \text{ldr } r0, [r1] \]

**Step 1**

- \( \text{ADDR} = r1 \)
- \( \text{DATA} = \text{Memory[ADDR]} \)
Load from Memory to Register (LDR)

Step 2  \( r0 = \text{DATA} \)
Store Register in Memory (STR)

\texttt{str \ r0, [r1]}

\textbf{Step 1} \hspace{1cm} \text{DATA} = \ r0
Store Register in Memory (STR)

str r0, [r1]

Step 2

ADDR = r1
Memory[ADDR] = DATA
Reset to continue editing code

```
1:  ldr  r0, =0x100
2:  mov  r1, #0xff
3:  str  r1, [r0]
4:  ldr  r2, [r0]
```
Turning on an LED
General-Purpose Input/Output (GPIO) Pins

54 GPIO Pins

(SDA1)
(SCL1)
(GPIO_GCLK)
(GPIO_GEN0)
(GPIO_GEN2)
(GPIO_GEN3)
(SPI_MOSI)
(SPI_MISO)
(SPI_SCLK)

5V

3V3

C64 100n 1005

C65 100n 1005

J8

1 2

3 4

5 6

7 8

9 10

11 12

13 14

15 16

17 18

19 20

21 22

23 24

25 26

27 28

29 30

31 32

33 34

35 36

37 38

39 40

40W 0.1" PIN HDR

(TXD0)
(RXDO)
(GPIO_GEN1)
(GPIO_GEN4)
(GPIO_GEN5)
(GPIO_GEN6)
(SPI_CE0_N)
(SPI_CE1_N)

54 GPIO Pins
Connect LED to GPIO 20

1 -> 3.3V
0 -> 0.0V (GND)
GPIO Pins are *Peripherals*

Peripherals are Controlled by Special Memory Locations

"Peripheral Registers"
Memory Map

Peripheral registers are mapped into address space

Memory-Mapped IO (MMIO)

MMIO space is above physical memory

Ref: BCM2835-ARM-Peripherals.pdf
# General-Purpose IO Function

**GPIO Pins can be configured to be INPUT, OUTPUT, or ALT0-5**

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Pin Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>The pin in an input</td>
</tr>
<tr>
<td>001</td>
<td>The pin is an output</td>
</tr>
<tr>
<td>100</td>
<td>The pin does alternate function 0</td>
</tr>
<tr>
<td>101</td>
<td>The pin does alternate function 1</td>
</tr>
<tr>
<td>110</td>
<td>The pin does alternate function 2</td>
</tr>
<tr>
<td>111</td>
<td>The pin does alternate function 3</td>
</tr>
<tr>
<td>011</td>
<td>The pin does alternate function 4</td>
</tr>
<tr>
<td>010</td>
<td>The pin does alternate function 5</td>
</tr>
</tbody>
</table>

3 bits required to select function
GPIO Function Select Register

"Function" is INPUT, OUTPUT (or ALT0-5)

8 functions requires 3 bits to specify

10 pins times 3 bits = 30 bits

32-bit register (2 wasted bits)

54 GPIOs pins requires 6 registers
### GPIO Function Select Registers Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Field Name</th>
<th>Description</th>
<th>Size</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 7E20 0000</td>
<td>GPFSEL0</td>
<td>GPIO Function Select 0</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0000</td>
<td>GPFSEL0</td>
<td>GPIO Function Select 0</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0004</td>
<td>GPFSEL1</td>
<td>GPIO Function Select 1</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0008</td>
<td>GPFSEL2</td>
<td>GPIO Function Select 2</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 000C</td>
<td>GPFSEL3</td>
<td>GPIO Function Select 3</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0010</td>
<td>GPFSEL4</td>
<td>GPIO Function Select 4</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0014</td>
<td>GPFSEL5</td>
<td>GPIO Function Select 5</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0018</td>
<td>-</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Watch out for …

Manual says: 0x7E200000
Replace 7E with 20: 0x20200000

Ref: BCM2835-ARM-Peripherals.pdf
# GPIO Pin Output Set Registers (GPSETn)

**Synopsis**
The output set registers are used to set a GPIO pin. The SET\{n\} field defines the respective GPIO pin to set, writing a “0” to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the SET\{n\} field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
<th>Type</th>
<th>Reset</th>
</tr>
</thead>
</table>
| 31-0   | SET\{n\} (n=0..31) | 0 = No effect  
1 = Set GPIO pin \(n\) | R/W   | 0     |

Table 6-8 – GPIO Output Set Register 0

<table>
<thead>
<tr>
<th>Bit(s)</th>
<th>Field Name</th>
<th>Description</th>
<th>Type</th>
<th>Reset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-22</td>
<td></td>
<td>Reserved</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>
| 21-0   | SET\{n\} (n=32..53) | 0 = No effect  
1 = Set GPIO pin \(n\). | R/W  | 0     |

Table 6-9 – GPIO Output Set Register 1
GPIO Function SET Register

20 20 00 1C : GPIO SET0 Register
20 20 00 20 : GPIO SET1 Register

Notes
1. 1 bit per GPIO pin
2. 54 pins requires 2 registers
// Set GPIO20 to be an output

// FSEL2 = 0x20200008
mov r0, #0x20 // r0 = #0x00000020
lsl r1, r0, #24 // r1 = #0x20000000
lsl r2, r0, #16 // r2 = #0x00200000
orr r1, r1, r2 // r1 = #0x20200000
orr r0, r1, #0x08 // r0 = #0x20200008

mov r1, #1 // 1 indicates OUTPUT
str r1, [r0] // store 1 to 0x20200008

Note this also makes GPIO 21-29 into inputs
Back to the ARM Instruction Set Architecture
3 Types of Instructions

1. Data processing instructions
2. Loads from and stores to memory
3. Conditional branches to new program locations
Data Processing Instructions and Machine Code
Figure 4-4: Data processing instructions

- **Destination register**
- **1st operand register**
- **Set condition codes**
  - 0 = do not alter condition codes
  - 1 = set condition codes
- **Operation Code**
  - 0000 = AND - Rd = Op1 AND Op2
  - 0001 = EOR - Rd = Op1 EOR Op2
  - 0010 = SUB - Rd = Op1 - Op2
  - 0011 = RSB - Rd = Op2 - Op1
  - 0100 = ADD - Rd = Op1 + Op2
  - 0101 = ADC - Rd = Op1 + Op2 + C
  - 0110 = SBC - Rd = Op1 - Op2 + C - 1
  - 0111 = RSC - Rd = Op2 - Op1 + C - 1
  - 1000 = TST - set condition codes on Op1 AND Op2
  - 1001 = TEQ - set condition codes on Op1 EOR Op2
  - 1010 = CMP - set condition codes on Op1 - Op2
  - 1011 = CMN - set condition codes on Op1 + Op2
  - 1100 = ORR - Rd = Op1 OR Op2
  - 1101 = MOV - Rd = Op2
  - 1110 = BIC - Rd = Op1 AND NOT Op2
  - 1111 = MVN - Rd = NOT Op2
- **Immediate Operand**
  - 0 = operand 2 is a register
  - 1 = operand 2 is an immediate value
- **Condition field**
  - 2nd operand register
  - shift applied to Rm
  - shift applied to Imm
  - Unsigned 8 bit immediate value

**Legend**
- **Cond**
- **00**
- **I**
- **OpCode**
- **S**
- **Rn**
- **Rd**
- **Operand 2**
data processing instruction

ra = rb op rc

Immediate mode instruction

Set condition codes

Data processing instruction

Always execute the instruction
<table>
<thead>
<tr>
<th>Assembly</th>
<th>Code</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0000</td>
<td>(ra = rb &amp; rc)</td>
</tr>
<tr>
<td>EOR (XOR)</td>
<td>0001</td>
<td>(ra = rb \oplus rc)</td>
</tr>
<tr>
<td>SUB</td>
<td>0010</td>
<td>(ra = rb - rc)</td>
</tr>
<tr>
<td>RSB</td>
<td>0011</td>
<td>(ra = rc - rb)</td>
</tr>
<tr>
<td>ADD</td>
<td>0100</td>
<td>(ra = rb + rc)</td>
</tr>
<tr>
<td>ADC</td>
<td>0101</td>
<td>(ra = rb + rc + CARRY)</td>
</tr>
<tr>
<td>SBC</td>
<td>0110</td>
<td>(ra = rb - rc + (1-CARRY))</td>
</tr>
<tr>
<td>RSC</td>
<td>0111</td>
<td>(ra = rc - rb + (1-CARRY))</td>
</tr>
<tr>
<td>TST</td>
<td>1000</td>
<td>(rb &amp; rc) (ra not set)</td>
</tr>
<tr>
<td>TEQ</td>
<td>1001</td>
<td>(rb \oplus rc) (ra not set)</td>
</tr>
<tr>
<td>CMP</td>
<td>1010</td>
<td>(rb - rc) (ra not set)</td>
</tr>
<tr>
<td>CMN</td>
<td>1011</td>
<td>(rb + rc) (ra not set)</td>
</tr>
<tr>
<td>ORR (OR)</td>
<td>1100</td>
<td>(ra = rb</td>
</tr>
<tr>
<td>MOV</td>
<td>1101</td>
<td>(ra = rc)</td>
</tr>
<tr>
<td>BIC</td>
<td>1110</td>
<td>(ra = rb &amp; \neg rc)</td>
</tr>
<tr>
<td>MVN</td>
<td>1111</td>
<td>(ra = \neg rc)</td>
</tr>
</tbody>
</table>
# data processing instruction
# ra = rb op rc
#

<table>
<thead>
<tr>
<th>op</th>
<th>rb</th>
<th>ra</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>00</td>
<td>i</td>
<td>oooo s</td>
</tr>
</tbody>
</table>

# i=0, s=0

<table>
<thead>
<tr>
<th>add</th>
<th>r1</th>
<th>r0</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>00</td>
<td>0</td>
<td>0100</td>
</tr>
</tbody>
</table>
# data processing instruction
# ra = rb op rc
#

<table>
<thead>
<tr>
<th>op</th>
<th>rb</th>
<th>ra</th>
<th>rc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 00 i</td>
<td>oooo s</td>
<td>bbbb aaaa</td>
<td>cccc cccc cccc</td>
</tr>
</tbody>
</table>

# i=0, s=0

<table>
<thead>
<tr>
<th>add</th>
<th>r1</th>
<th>r0</th>
<th>r2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110 00 0 0100 0 0001 0000 0000 0000 0010</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1110 0000 1000 0001 0000 0000 0000 0000 0010
E 0 8 1 0 0 0 0 2
**least-significant-byte (LSB)**

- E0
- 81
- 00
- 02

**most-significant-byte (MSB)**

<table>
<thead>
<tr>
<th>ADDR+3</th>
<th>E0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR+2</td>
<td>81</td>
</tr>
<tr>
<td>ADDR+1</td>
<td>00</td>
</tr>
<tr>
<td>ADDR</td>
<td>02</td>
</tr>
</tbody>
</table>

**little-endian**

(Least Significant Byte, LSB, first)

**ARM uses little-endian**
most-significant-byte (MSB)

least-significant-byte (LSB)

big-endian (MSB first)
# data processing instruction
# ra = rb op #imm
# #imm = uuuu uuuu

```
add     r1     r0     imm
1110 00 1 0100 0 0001 0000 0000 uuuu uuuu
```

```
add r0, r1, #1
```

```
add     r1     r0     #1
1110 00 1 0100 0 0001 0000 0000 0000 0000 0001
```
# data processing instruction
# ra = rb op #imm
# #imm = uuuu uuuu

```
add  r1  r0  imm
1110 00 1 0100 0 0001 0000 0000 0001 uuuu uuuu
```

add r0, r1, #1
```
add  r1  r0  #1
1110 00 1 0100 0 0001 0000 0000 0001 0000 0001
```

```
1110 0010 1000 0001 0000 0000 0000 0000 0001 0001
E 2 8 1 0 0 0 0 1
```
# data processing instruction
# ra = rb op imm
# imm = (u0000 u0000) ROR (2*rrrr)

```
1110 00 1 0000 0 bbbb aaaa rrrr uuuu uuuu
```

ROR means *Rotate Right* (imm>>>rotate)
# data processing instruction
# ra = rb op imm
# imm = (uuuu uuuu) ROR (2*rrrr)

1110 00 1 oooo 0 bbbb aaaa rrrr uuuu uuuu

add r0, r1, #0x10000

1110 00 1 0100 0 0001 0000 1000 0000 0001

0x01>>>16
0000 0000 0000 0000 0000 0000 0000 0000 0001
0000 0000 0000 0001 0000 0000 0000 0000 0000
# data processing instruction
# \( ra = rb \text{ op imm} \)
# \( \text{imm} = (\text{uuuuu uuuuu}) \text{ ROR (2*rrrr)} \)

```
op       rb       ra       ror       imm
1110 00 1 ooooo 0 bbbbaaaa rrrruuuuuuu
```

add \( r0, r1, \#0x10000 \)
```
add       r1       r0       0x01>>>2*8
1110 00 1 0100 0 0001 0000 1000 0000 0001
```

```
1110 0010 1000 0001 0000 1000 0000 0000 0001
E   2    8    1   0   8   0   1
```
```assembly
/// SET0 = 0x2020001c
mov r0, #0x20          // r0 = 0x00000020
lsrl r1, r0, #24       // r1 = 0x20000000
lsrl r2, r0, #16       // r2 = 0x00200000
orr r0, r1, r2         // r0 = 0x20200000
orr r0, r0, #0x1c      // r0 = 0x2020001c

/// SET0 = 0x2020001c
mov r0, #0x2000000000   // 0x20>>>8
orr r0, #0x0020000000   // 0x20>>>16
orr r0, #0x00000001c    // 0x1c>>>0
```
```assembly
/// SET0 = 0x2020001c
mov r0, #0x20 // r0 = 0x00000020
lsl r1, r0, #24 // r1 = 0x20000000
lsl r2, r0, #16 // r2 = 0x00200000
orr r0, r1, r2 // r0 = 0x20200000
orr r0, r0, #0x1c // r0 = 0x2020001c

// SET0 = 0x2020001c
mov r0, #0x20000000 // 0x20>>>8
orr r0, #0x00200000 // 0x20>>>16
orr r0, #0x0000001c // 0x1c>>>0

Using the barrel shifter lets us make the code 40% shorter (and 40% faster)
```
Load from Memory to Register (LDR)

```assembly
ldr r0, [r1, #4]
```

**ADDR = r1 + 4**

**DATA = Memory[ADDR]**
// configure GPIO 20 for output
ldr r0, [pc + 20]
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, [pc + 12]
mov r1, #0x00100000
str r1, [r0]

loop: b loop

.word 0x20200008
.word 0x2020001C
// configure GPIO 20 for output
ldr r0, =0x20200008
mov r1, #1
str r1, [r0]

// set bit 20
ldr r0, =0x2020001C
mov r1, #0x00100000
str r1, [r0]

loop: b loop
3 steps to run an instruction

Fetch  Decode  Execute
3 instructions takes 9 steps
To speed things up, steps are overlapped ("pipelined")
To speed things up, steps are overlapped ("pipelined")

PC value in the executing instruction is equal to the pc value of the instruction being fetched - which is 2 instructions ahead (PC+8)
Blink
mov r1, #(1<<20)

// Turn on LED connected to GPIO20
ldr r0, SET0
str r1, [r0]

// Turn off LED connected to GPIO20
ldr r0, CLR0
str r1, [r0]
// Configure GPIO 20 for OUTPUT

loop:

  // Turn on LED

  // Turn off LED

  b loop
Loops and Condition Codes
```assembly
// define constant
.equ DELAY, 0x3f0000

mov r2, #DELAY

loop:
    subs r2, r2, #1  // s set cond code
    bne loop        // branch if r2 != 0
```
Orthogonal Instructions

Any operation

Register vs. immediate operands

All registers the same**

Predicated/conditional execution

Set or not set condition code

Orthogonality leads to composability
Further Reading

If you want to learn more about high-level computer organization and instructions, Chapter 2 of Computer Organization and Design: The Hardware/Software Interface (Patterson and Hennessy) is an excellent place to start.

Or take EE180 in Spring!
The Fun Begins ...

Lab 1

- Install tool chain before lab
- Read lab 1 instructions (now online)
- Assemble Raspberry Pi Kit
- Bring USB-C to USB-A adapter (if you need it)

Assignment 1

- Larson scanner
- YEAH office hours Thu 3-4pm in B21
Definitive References

BCM2835 peripherals document + errata

Raspberry Pi schematic

ARM11/ARMv6 reference manual

see Resources on cs107e.github.io
Extra Material on Branches
Branch Instructions
Condition Codes

Z - Result is 0

N - Result is <0

C - Carry generated

V - Arithmetic overflow

*Carry and overflow will be covered later*
<table>
<thead>
<tr>
<th>Code</th>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Z set</td>
<td>equal</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Z clear</td>
<td>not equal</td>
</tr>
<tr>
<td>0010</td>
<td>CS</td>
<td>C set</td>
<td>unsigned higher or same</td>
</tr>
<tr>
<td>0011</td>
<td>CC</td>
<td>C clear</td>
<td>unsigned lower</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>N set</td>
<td>negative</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>N clear</td>
<td>positive or zero</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>V set</td>
<td>overflow</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>V clear</td>
<td>no overflow</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>C set and Z clear</td>
<td>unsigned higher</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>C clear or Z set</td>
<td>unsigned lower or same</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>N equals V</td>
<td>greater or equal</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>N not equal to V</td>
<td>less than</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Z clear AND (N equals V)</td>
<td>greater than</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Z set OR (N not equal to V)</td>
<td>less than or equal</td>
</tr>
<tr>
<td>1110</td>
<td>AL</td>
<td>(ignored)</td>
<td>always</td>
</tr>
</tbody>
</table>
# branch

<table>
<thead>
<tr>
<th>cond</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>cccc</td>
<td>101L</td>
</tr>
<tr>
<td></td>
<td>o000 o000 o000 o000 o000 o000 o000 o000</td>
</tr>
</tbody>
</table>

b = bal = branch always

<table>
<thead>
<tr>
<th>cond</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>101L</td>
</tr>
<tr>
<td></td>
<td>o000 o000 o000 o000 o000 o000 o000 o000</td>
</tr>
</tbody>
</table>

bne

<table>
<thead>
<tr>
<th>cond</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>101L</td>
</tr>
<tr>
<td></td>
<td>o000 o000 o000 o000 o000 o000 o000 o000</td>
</tr>
</tbody>
</table>