Module 1

Microcontroller Programming Techniques

Textbook:

Chapter 1 (p1-26)

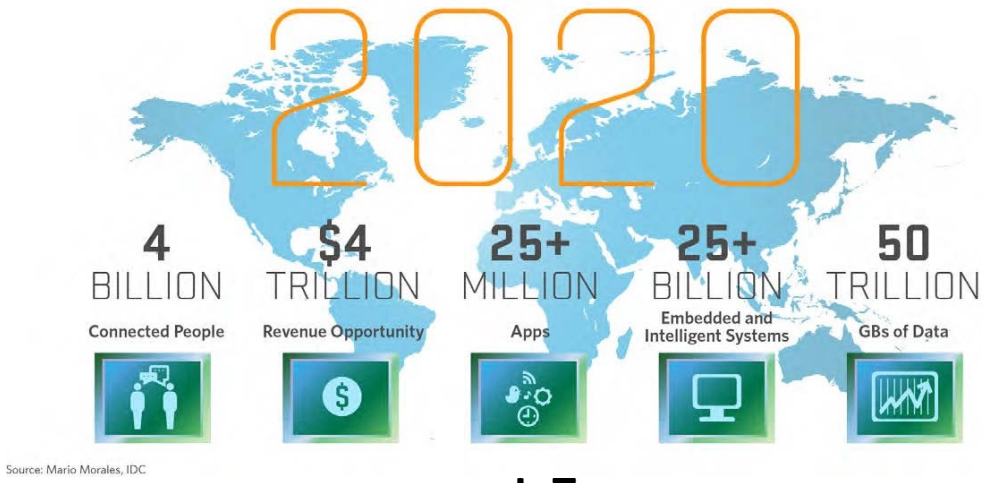
Chapter 2 (p27-54): Mostly review from 270

Tim Rogers 2022

Learning Outcome #1

"An ability to program a microcontroller to perform various tasks"

Why?



IoT

Embedded systems in cars

Learning Outcome #1

"An ability to program a microcontroller to perform various tasks"

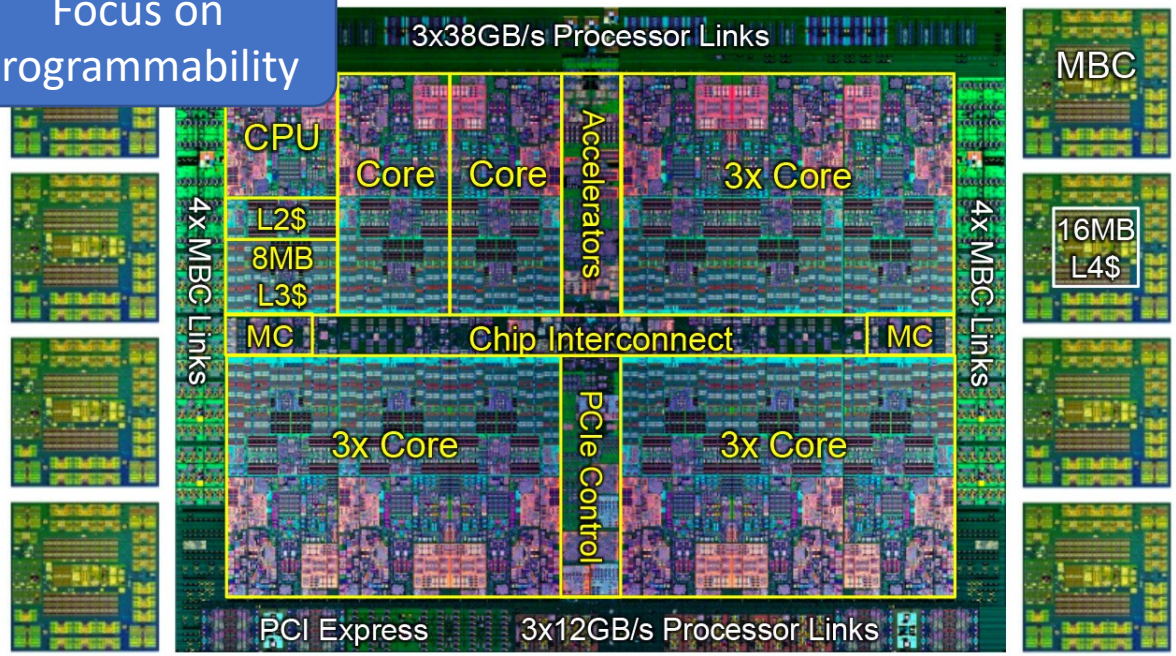
How?

A. Architecture and Programming Model
B. Instruction Set Basics
C. Addressing modes and conditionals
D. Practical Assembly Language Programming
E. Stacks and Recursive Functions*

What makes a Microcontroller?

General Purpose

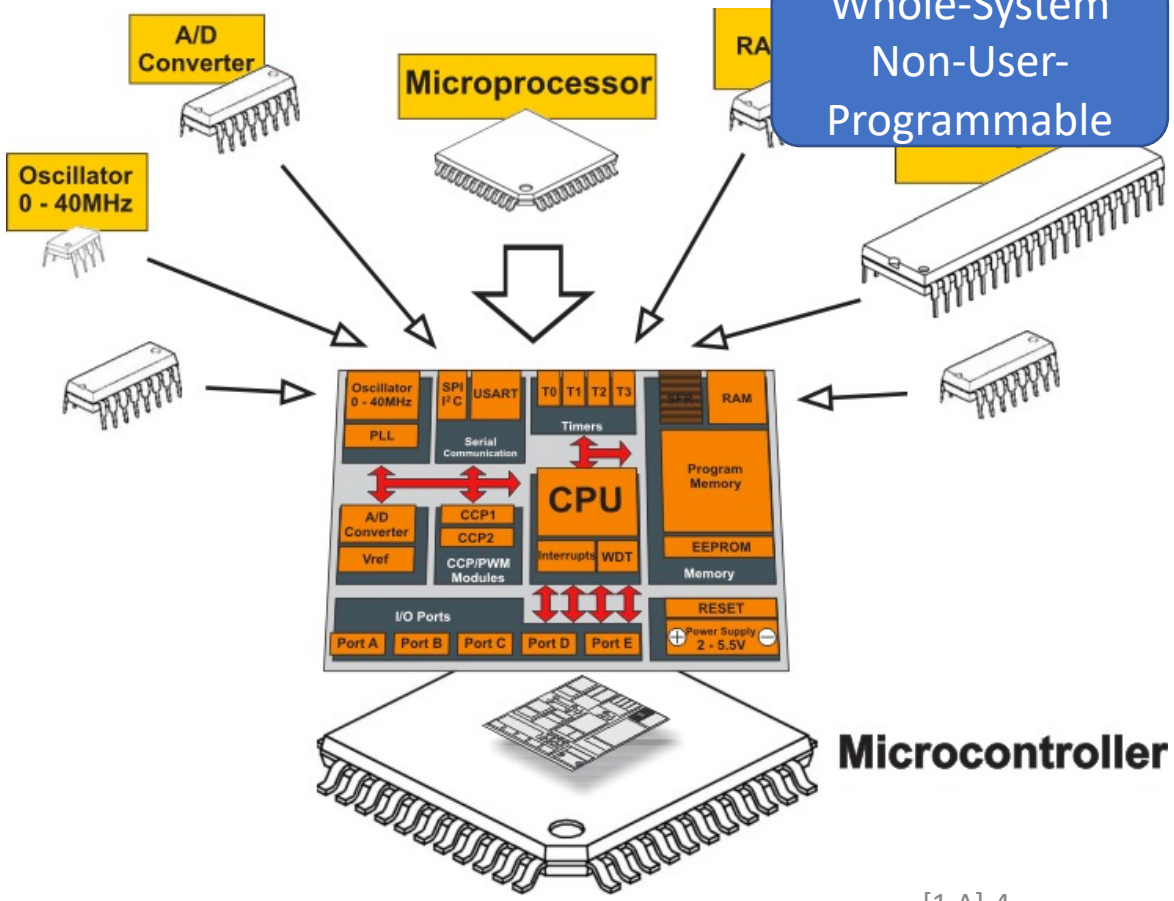
Focus on Programmability



IBM Power 8 Microprocessor

Embedded

Whole-System Non-User-Programmable



Microcontroller

Different Objectives

General Purpose

- High-performance
- Need to run “anything” relatively well
- Less energy constrained
- “Limitless” Memory
- Silicon devoted to compute and caches

Embedded

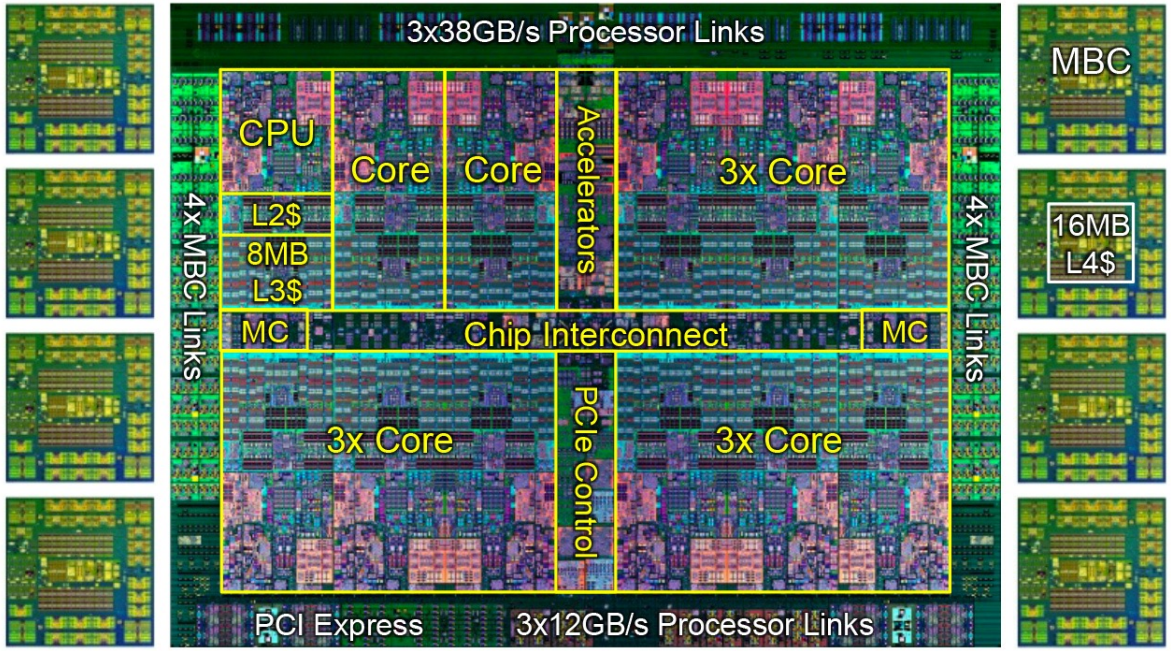
- Limited ability
- Inexpensive
- Limited memory
- Low-energy
- Silicon devoted to many peripherals (although core still consumes a lot)

Characteristics of general-purpose processors

- Virtual memory (Page Tables, TLBs)
- Deep cache hierarchy
- Large number of registers
- Hardware floating point
- Deep Pipelines/ Out of Order Exec.
- Complex Prediction Mechanisms
- Multicore

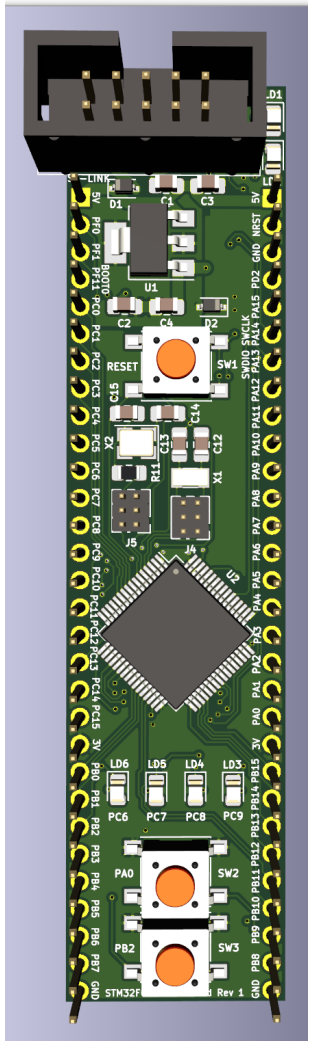
Interrupts mostly get in the way

Take ECE 437!



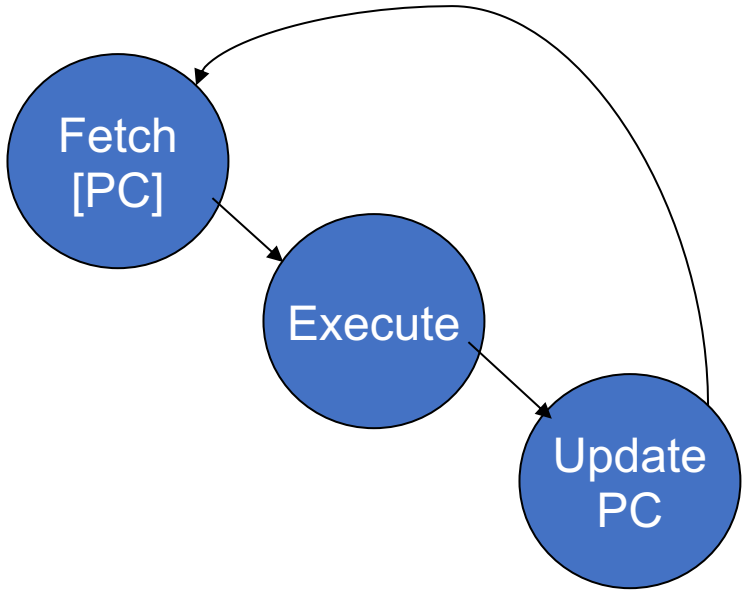
Characteristics of embedded processors

- Live and die by interrupts!
- Few registers (need to switch contexts fast)
- Circuitry devoted to analog
- Often rely on hand-tuned assembly code



This basics of all computers

- What does it do?
  - Fetch instruction from address in Program Counter (PC)
  - Execute instruction
  - Update PC to point to next instruction



The Basics

- C statement
- **f = (g + h) - (i + j)**

What does the CPU need to do?

What assembly instructions might look like

- Assembly instructions*

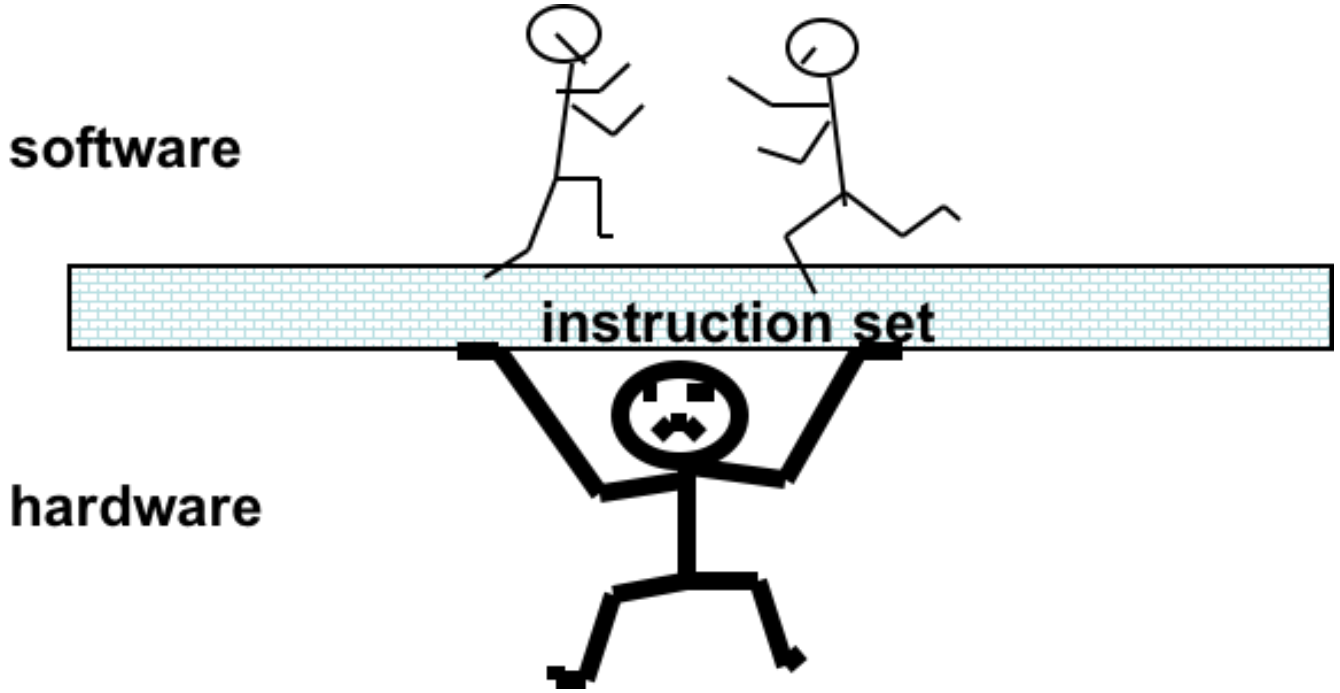
```
add t0, g, h   // t0 = g + h
add t1, i, j   //  t1 = i + j
sub f, t0, t1  //  f  = t0 – t1
```

- Opcodes/mnemonic, operands, source/destination

Not the only of doing business!

Instruction Set Architectures (ISAs)

- Set of operations **visible to the programmer**
- ISA is the contract between the SW and HW



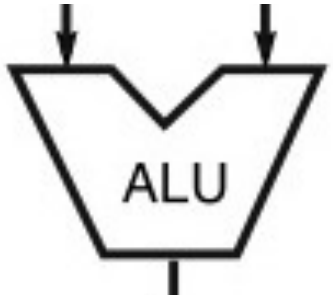
- Examples: CPU12, x86, PowerPC, **ARM**, SPARC, etc…

Different Types of ISAs

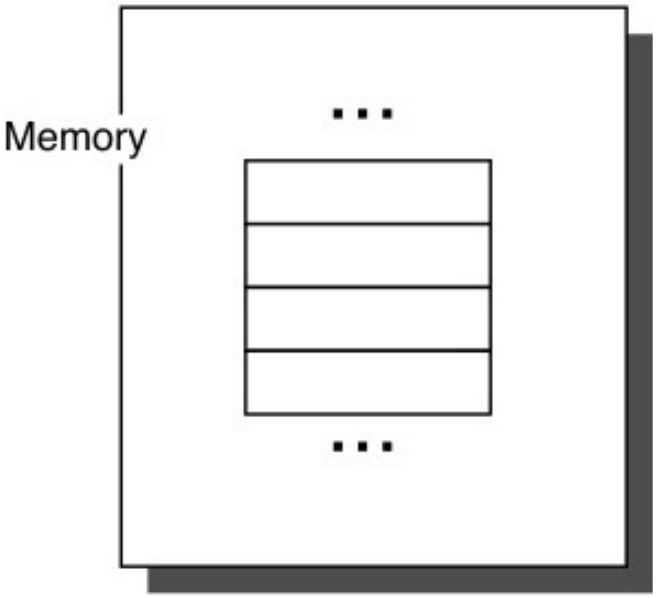
- CISC (Complex Instruction Set Computer)
- RISC (Reduced Instruction Set Computer)
- DSPs (Digital Signal Processors)

ARM Cortex-M0 (32-bits)

ISA defines
Interaction between processor and memory

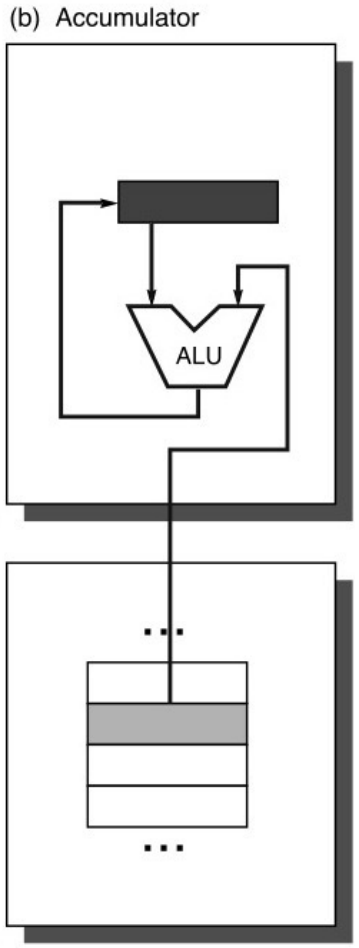


??



Some common Architectures

How to compute "Z = X + Y":

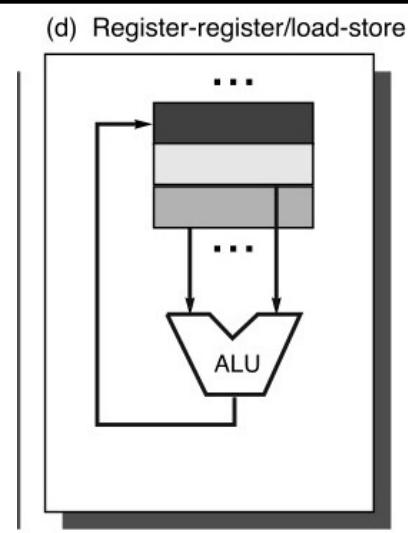


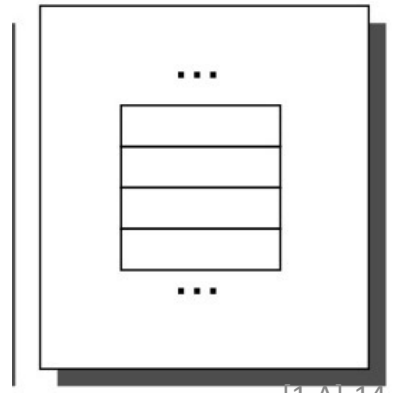
Accumulator

```
LoadA   X
AddA    Y
StoreA  Z
```

Reg/Reg

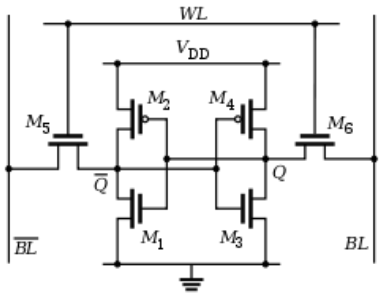
```
Load    R1,X
Load    R2,Y
Add     R3,R1,R2
Store   R3,Z
```





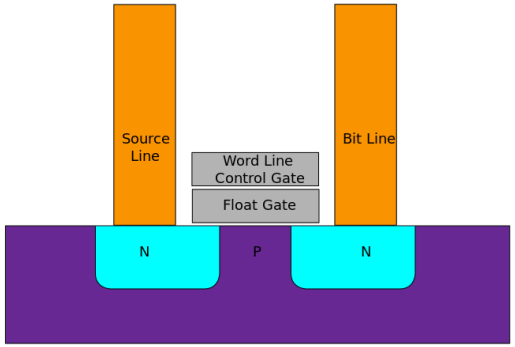
ARM Cortex-M0 (ECE 362)

Memory Types



SRAM (Static Random Access Memory)

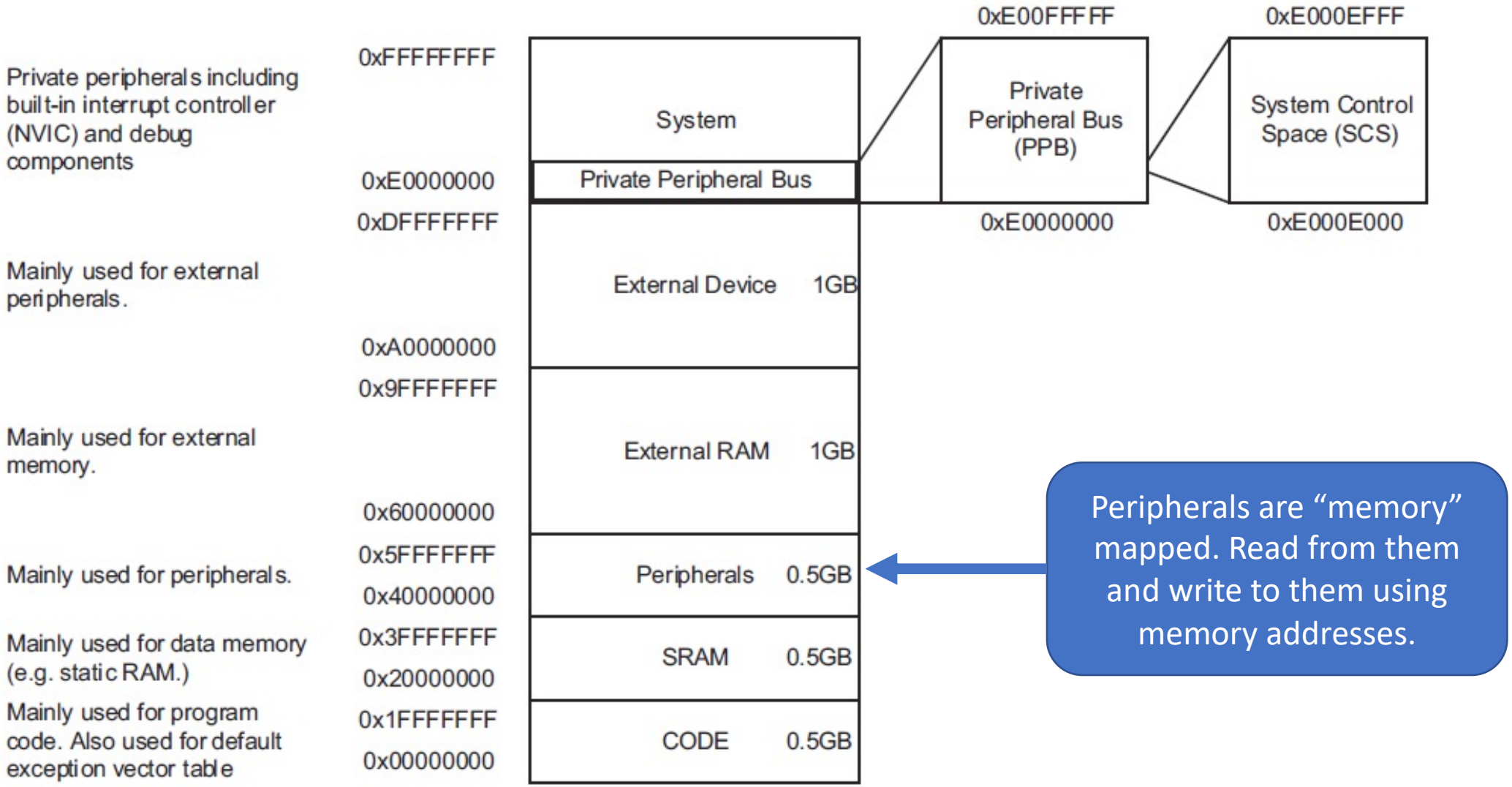
- Volatile
  - Variables
  - Stack
  - Buffers
  - Test Code



Flash Memory

- Non-Volatile
  - App Code
  - Static Data
  - Vectors (reset and interrupts)

Memory Map


0xFFFFFFFF
Private peripherals including built-in interrupt controller (NVIC) and debug components
System
0xE00FFFFF
0xE000EFFF
Private Peripheral Bus (PPB)
System Control Space (SCS)
0xE0000000
Private Peripheral Bus
0xDFFFFFFF
0xE0000000
0xE000E000
Mainly used for external peripherals.
External Device 1GB
0xA0000000
0x9FFFFFFF
Mainly used for external memory.
External RAM 1GB
0x60000000
0x5FFFFFFF
Mainly used for peripherals.
Peripherals 0.5GB
0x40000000
0x3FFFFFFF
Mainly used for data memory (e.g. static RAM.)
SRAM 0.5GB
0x20000000
0x1FFFFFFF
Mainly used for program code. Also used for default exception vector table
CODE 0.5GB
0x00000000
Peripherals are “memory” mapped. Read from them and write to them using memory addresses.

Microcontroller Breakdown

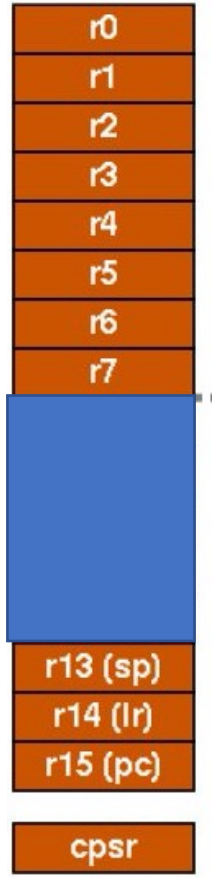
- 32-bit ARM Cortex-M0 CPU.
- 8MHz clock multiplied x6 by a phase-locked loop (PLL) to get 48 MHz.
- 3-stage pipeline. Most instructions take 3 cycles, but a new instruction starts each cycle.
- Built-in Flash ROM to store program and SRAM to hold read/write data.
- 32 maskable interrupt channels w/ 4 priority levels
- 12-bit ADC, 12-bit DAC
- SPI, I2S, I2C, USART, CAN, HDMI CEC peripherals
- Several timers with integrated PWM
- So much more:
- https://engineering.purdue.edu/ece362/refs/

ARM Cortex-M0 CPU: Our Embedded CPU

- Maximum power dissipation: 400 – 500mW
- No heatsink needed. Easy to “embed.”
- Sometimes people try to embed a conventional high-performance CPU.
  - This kind of thing needs a heatsink, fan, ventilation, etc.
- By comparison, laptop CPUs use ~50W TDP, data center CPUs ~200W, Discrete GPUs >250W!
- Not good for embedded use.

Thumb ISA

- Used to improve code density
  - Cannot access r8 through r12
  - Implements fewer op-codes
- Still operate on 32-bit registers



Thumb Encoding

Encoding T1 All versions of the Thumb ISA.

```
ADDS <Rd>,<Rn>,#<imm3>
ADD<c> <Rd>,<Rn>,#<imm3>
```

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		

Encoding T2 All versions of the Thumb ISA.

```
ADDS <Rdn>,#<imm8>
ADD<c> <Rdn>,#<imm8>
```

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

Encoding T3 ARMv7-M

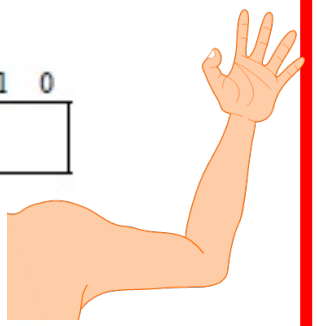
```
ADD{S}<c>.W <Rd>,<Rn>,#<const>
```

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							

Encoding T4 ARMv7-M

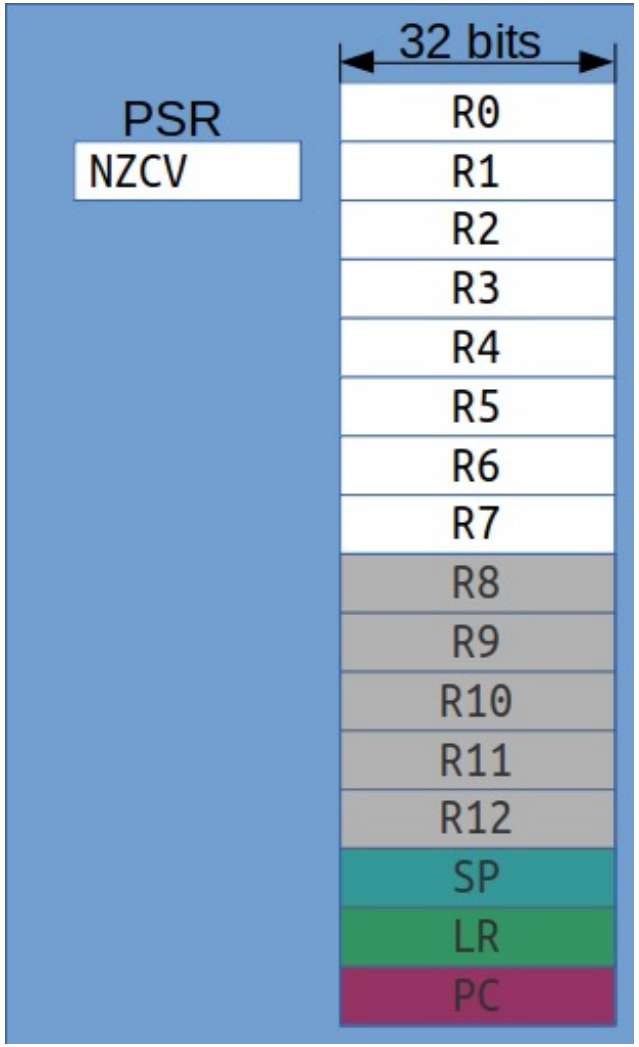
```
ADDW<c> <Rd>,<Rn>,#<imm12>
```

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							



Our ISA's Registers

- 16 registers
  - R15 is the PC.
  - R13 is the SP.
  - R14 is the "Link Register" (LR)
  - Most instructions can only use R0 – R7.
- Program Status Register (PSR).
  - Upper four bits are the flags.

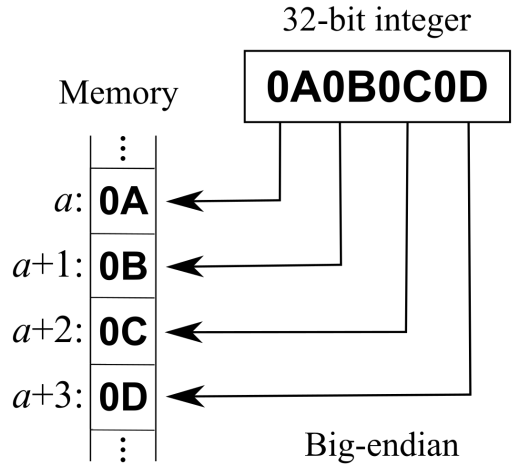


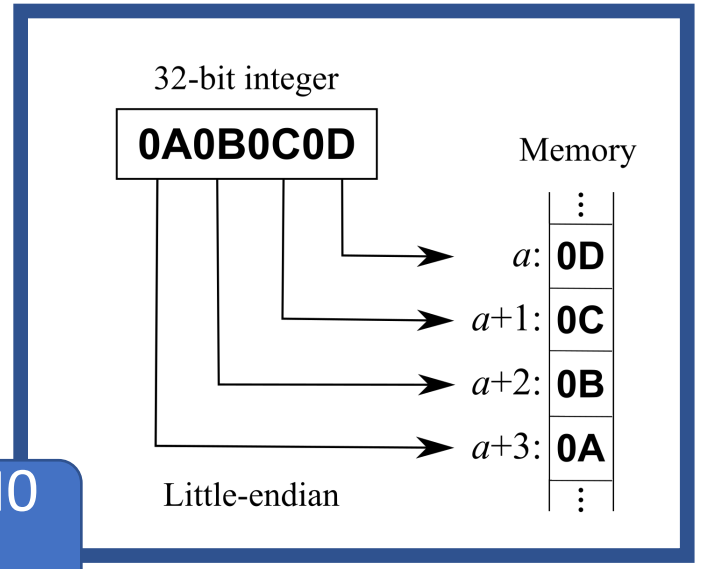
Upper PSR Bits

- "N" – "negative" flag (most significant bit (sign) of computation)
- "Z" – "zero" flag (set if result of computation is zero)
- "C" – "carry" flag. (set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition)
- "V" – "overflow" flag (set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition)

Endianness

- Word storage
- Word = bytes (0x400,0x401,0x402,0x403)?
- Word = bytes (0x403,0x402,0x401,0x400)?
  - It depends…
- Big endian: MS Byte at address xxxxxx00b
  - e.g., IBM, SPARC
- Little endian: MS Byte at address xxxxxx11b
  - e.g., Intel x86
- Mode selectable
  - e.g., PowerPC, MIPS,ARM (generally)





ARM Cortex-M0 (ECE 362)

Instruction Representation

- Just a bunch of bits
- Remember? Simple Computer?

Opcode	Mnemonic	Function Performed
0 0 0	LDA *addr*	Load A with contents of location *addr*
0 0 1	STA *addr*	Store contents of A at location *addr*
0 1 0	ADD *addr*	Add contents of *addr* to contents of A

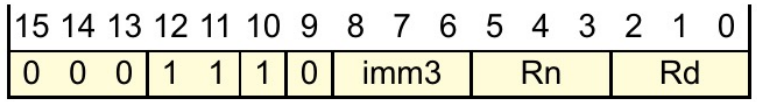
Example Encoding for Our Instructions

A6.7.2 ADD (immediate)

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

Encoding T1 All versions of the Thumb instruction set.

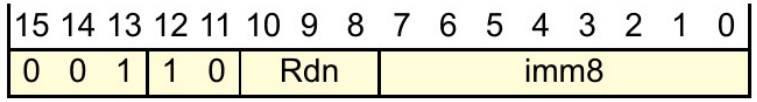
```
ADDS <Rd>,<Rn>,#<imm3>
```




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

Encoding T2 All versions of the Thumb instruction set.

```
ADDS <Rdn>,#<imm8>
```




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

General Encoding format

A5.2 16-bit Thumb instruction encoding

The encoding of 16-bit Thumb instructions is:

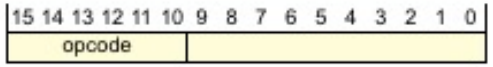


Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-85
010000	*Data processing* on page A5-86
010001	*Special data instructions and branch and exchange* on page A5-87
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-141
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-88
10100x	Generate PC-relative address, see *ADR* on page A6-115
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-111
1011xx	*Miscellaneous 16-bit instructions* on page A5-89
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A6-175
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A6-137
1101xx	*Conditional branch, and Supervisor Call* on page A5-90
11100x	Unconditional Branch, see *B* on page A6-119

Addressing Modes

- Definition: The CPU uses an *addressing mode* to determine how instructions get operands.
- When getting those operands from memory, it specifies how the address is computed.

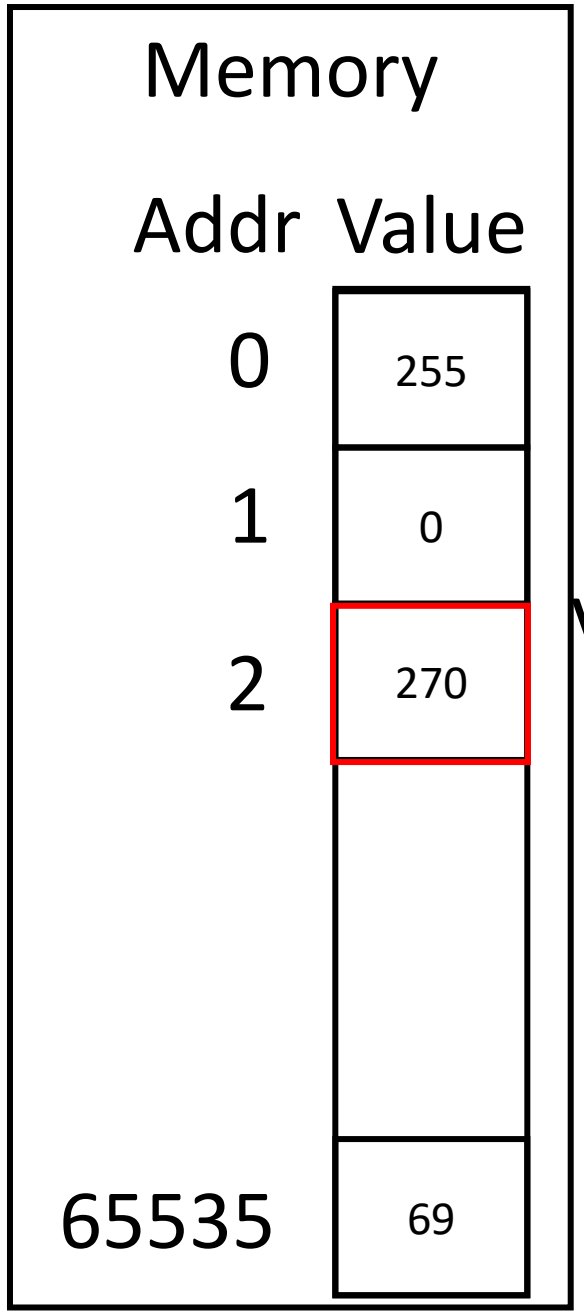
```
# C-code
f = (g + h) - (i + j)

# "assembly"
add t0, g, h   // t0 = g + h
add t1, i, j   //  t1 = i + j
sub f, t0, t1  //  f  = t0 – t1
```

g,h,i,j and f all need to be a 32-bit address. Addressing modes tell us how we compute that address.

Addressing Modes

- Purpose:
  - Need to access memory
  - Need an ***ADDRESS***

??



Value not possible

Simple Computer Addressing Modes

The Simple Computer had 3 addressing modes:

- **Register**: Operand is read from a register
  - ADD R6,R11
- **Immediate**: Operand is encoded directly in the instruction:
  - ADDI R5,#0f
- **Absolute**: Operand referred to a specific memory location.
  - LDL R3,01fc

Cortex-M0 ISA

- RISC with compromises
  - Most instructions can refer only to registers R0 – R7.
  - Only small constants are used in instructions.
  - Load/Store architecture: Memory references separate from arithmetic.
- Four major addressing modes:
  - **Immediate**: small value contained in instruction
  - **Register**: src/dst register in instruction
  - **Offset**: value is at an address that's a relative (constant) offset from a register
  - **Indexed**: value is at address specified by register plus a second register

Accessing Memory with Loads and Stores

- The Cortex-M0 CPU cannot reference memory with arithmetic or logical instructions.
- It can only reference memory with load and store instructions.
  - That’s why it’s called a load/store architecture.
- It also cannot refer to an absolute address.
  - Instead, it uses an address stored in a register.
  - A load or store is relative to one of registers R0 – R7.

The LDR (Load Register) Instruction

- Let's say R0 holds the value 0x20000000.
- We want to load the four-byte value contained in addresses
- 0x20000000 – 0x20000003 into register R1.
- This is the instruction:
  - LDR R1, [R0]
- "Use the value in R0 as an address. Read 4-bytes starting from that address, and put the result in R1."
- Caveat: R0 must be evenly divisible by 4.
  - If not, it will invoke a fault handler. You don't want that.

The STR (Store Register) Instruction

- Let's say R0 holds the value 0x20000000.
- We want to store the four-byte value contained in R1 to the addresses 0x20000000 – 0x20000003.
- This is the instruction:
  - STR R1, [R0]
- "Use the value in R0 as an address. Write the value of R1 into the 4-bytes starting at that address."
- Caveat: R0 must be evenly divisible by 4.
  - If not, it will invoke a fault handler. You don't want that.

```
# C-code
f = (g + h)

# "assembly"
add f, g, h
```

```
# Cortex M-0
# Assume R0 holds 0x0 (g)
#        R1 holds 0x4 (h)
#        R2 holds 0x8 (f)

LDR R3, [R0]
LDR R4, [R1]
ADDS R5, R3, R4
STR R5, [R2]
```