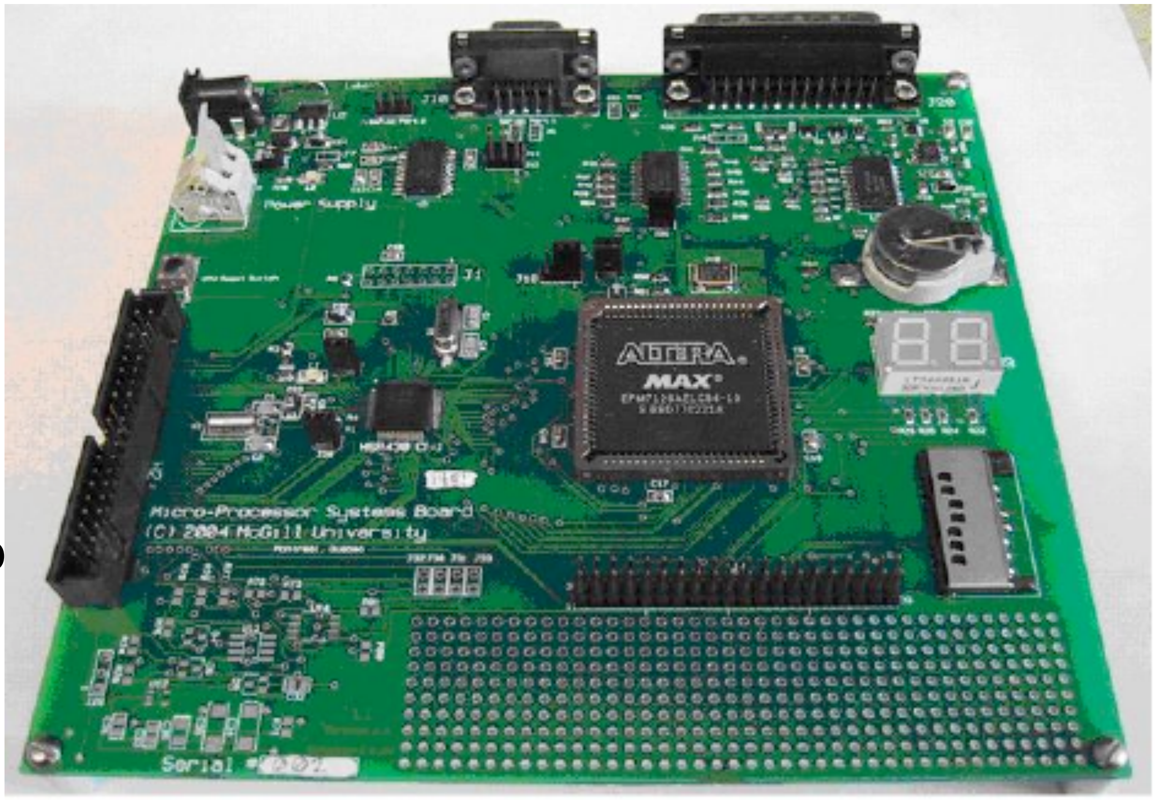
# ECSE 426
# ISA, Assemblers and Labs

Zeljko Zilic

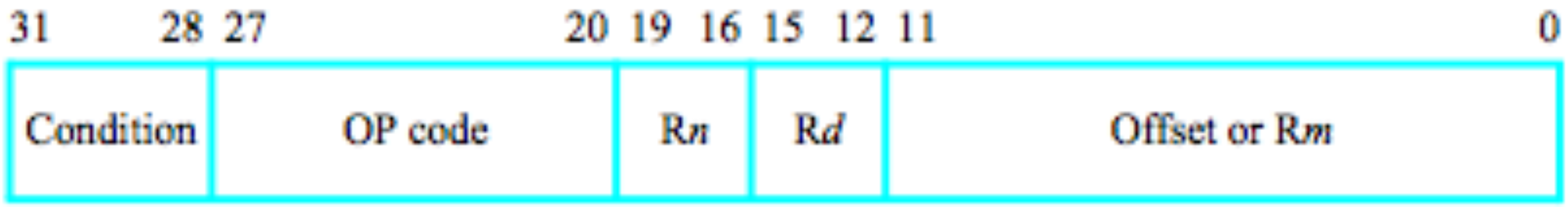
Room 546
McConnell Building
zeljko@ece.mcgill.ca
www.macs.ece.mcgill.ca/~zeljko



## ISA – Machine Code

- Machine code vs. assembly
  - Machine code is a string of bits that processors "read". It is practically unreadable for humans
  - Assembly is intended for human reading
- In assembler the following instruction formatting is commonly used:

```
  label: opcode operand1, operand2, … comments
```



## Assembly Language

```
Label: opcode operand1, operand2, … comments
```

- Label is optional
- Opcode is the actual instruction
- Depending on the instruction there can be two or mode operands
  - Usually, the first operand is the destination of the operation

## ARM Cortex ISA: ARMv7-M

- Instructions:
  - Data movement: single and multiple word
  - Data processing: arithmetic, logic, shift, rotate
  - Branches: conditional, subroutine, table (case)
  - State change: int. enable, spec. reg., ITE, SVC
  - Semaphores, Barrier, Hint, Prefetch
  - Coprocessor, floating-point, misc.

# Addressing Modes

- Addressing mode describes manner in which data in CPU registers and memory is accessed
  - During program execution data can reside either in:
    - Machine instruction itself
      - Displacement
    - CPU registers
    - Main memory
    - Secondary storage
      - Access to this type of data not direct
- Three addressing modes are generally supported by assembly language:
  - Data register direct addressing
  - Address register direct addressing
  - Address register indirect with displacement addressing
- Accessing data
  - When data ***is*** part of instruction then it can be accessed immediately
    - Immediate data (addressing scheme)
  - When data ***is not*** part of instruction then we have three classes of data access:
    - Register addressing scheme
    - Memory addressing scheme

# Instruction Operands

- Instruction operand can be:
  - ARM register
  - Constant
  - Instruction-specific parameter
- Many instructions are two-operand
  - Many data-processing instructions have flexible second operand `Operand2`, which can be
    - Constant
    - Register

## `Operand2` as a Constant

- Constant `Operand2` is specified in form:
  - `#constant`
  - Where `#constant` can be
    - Any constant that can be produced by shifting 8-bit value left by any number of bits within a 32-bit word
    - Any constant of the form 0x00XY00XY
    - Any constant of the form 0xXY00XY00
    - Any constant of the form 0xXYXYXYXY
      - X and Y are hex numbers

## Load (LDR) and Store (STR)

- LDR instruction loads register with value from memory
- STR instruction store register value into memory
- Memory address to load from or to store to is an offset from register Rn
  - Offset is specified by register Rm
- Value to load or store can be a byte, halfword or word

## ISA: Data Movement Instructions

Load, Store and Move to register

```
LDR Rd, EA //EA=effective address
STR Rd, EA
MOV Rd, Rm
```

Major Variations:

- MVN: Move negative
- LDM, STM: Load and Store multiple
- PUSH, POP: stack (shortcut for LDM, STM via SP)
- Conditional; doubleword; exclusive

## EA: Basic Addressing Mode

- Pre-indexed – effective address (EA) of operand: sum of base register and signed offset
  - Operator "[]" denotes indirection in reference (to memory)
- Example:

```
LDR Rd, [Rn, #offset]
```

```
Effect: Rd <-[[Rn]+offset]
```

```
LDR Rd, [Rn, Rm]
```

```
Effect: Rd<-[[Rn]+[Rm]]
```

## Store Timing

`STR Rd, [Rn, #offset]`

`Effect: Rd ->[[Rn]+offset]`

- Execution takes always one cycle
  - Address generation is performed in the initial cycle, and data store is performed at the same time as the next instruction is executing

# Rationale for Addressing Modes

- Instructions have fixed length, and they have to accommodate different addressing modes with different length of addresses
- EA derived from register Rn and offset offs
- Offset: register Rm or immediate value #imm
- Indexed:
  - Pre: offset added to index reg. before op [Rn, offs]
  - Post: after op, reg. augmented for offset [Rn], offs
  - Pre with writeback: pre-ind. and update register [Rn, offs]!
- Relative to PC: EA=[PC]+#imm
- Also: inline shifting offset register - barrel shifter
- Example (ARM, no Thumb/Thumb2): `LDR R4, [R2, -R3, LSL #4]`**!**

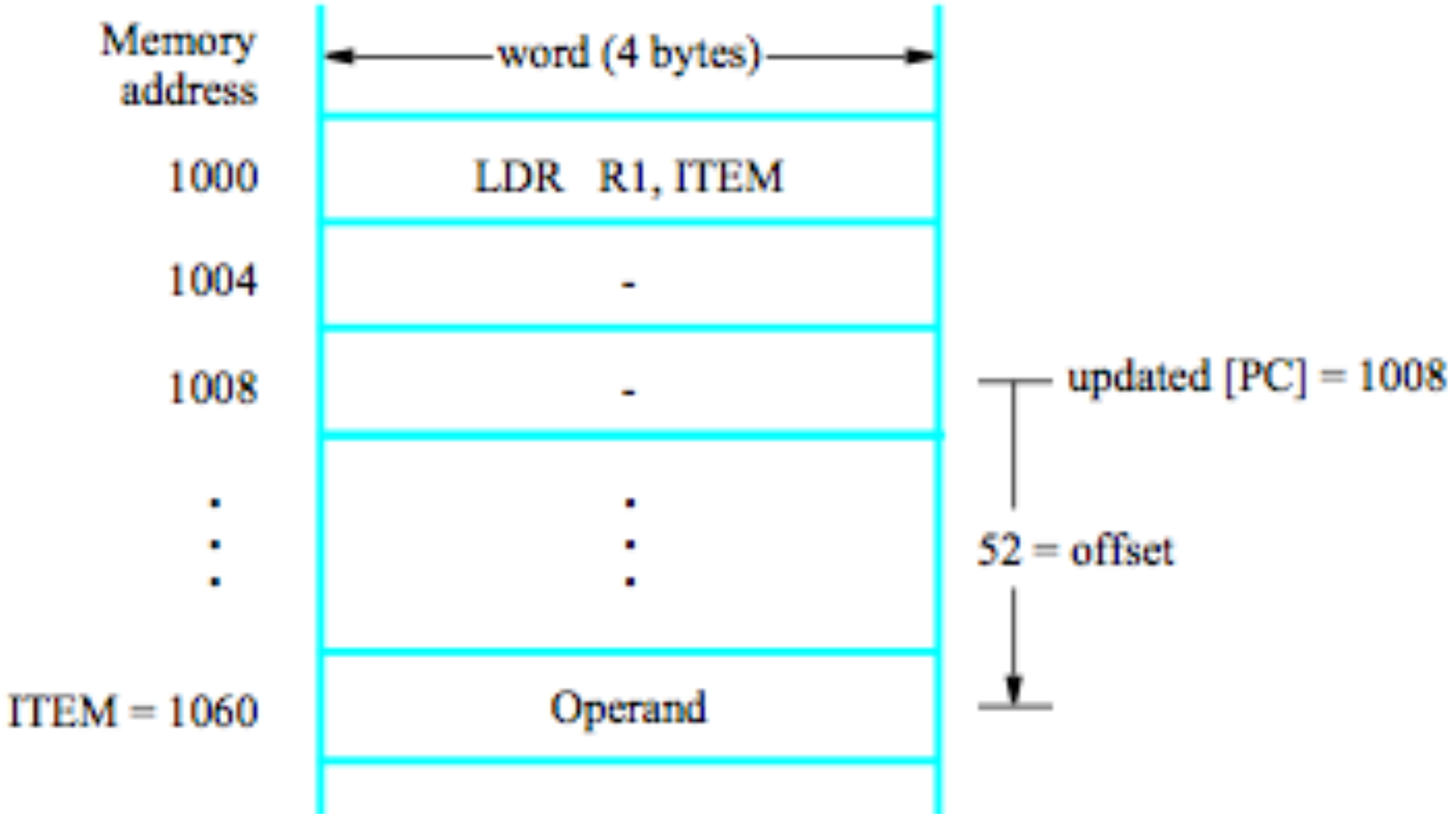
  Result: `R4 <- [[R2]-16x[R3]];` main operation

  `R2 <- R2-16xR3` ; index reg. update
- Summary of [HVZM11] textbook

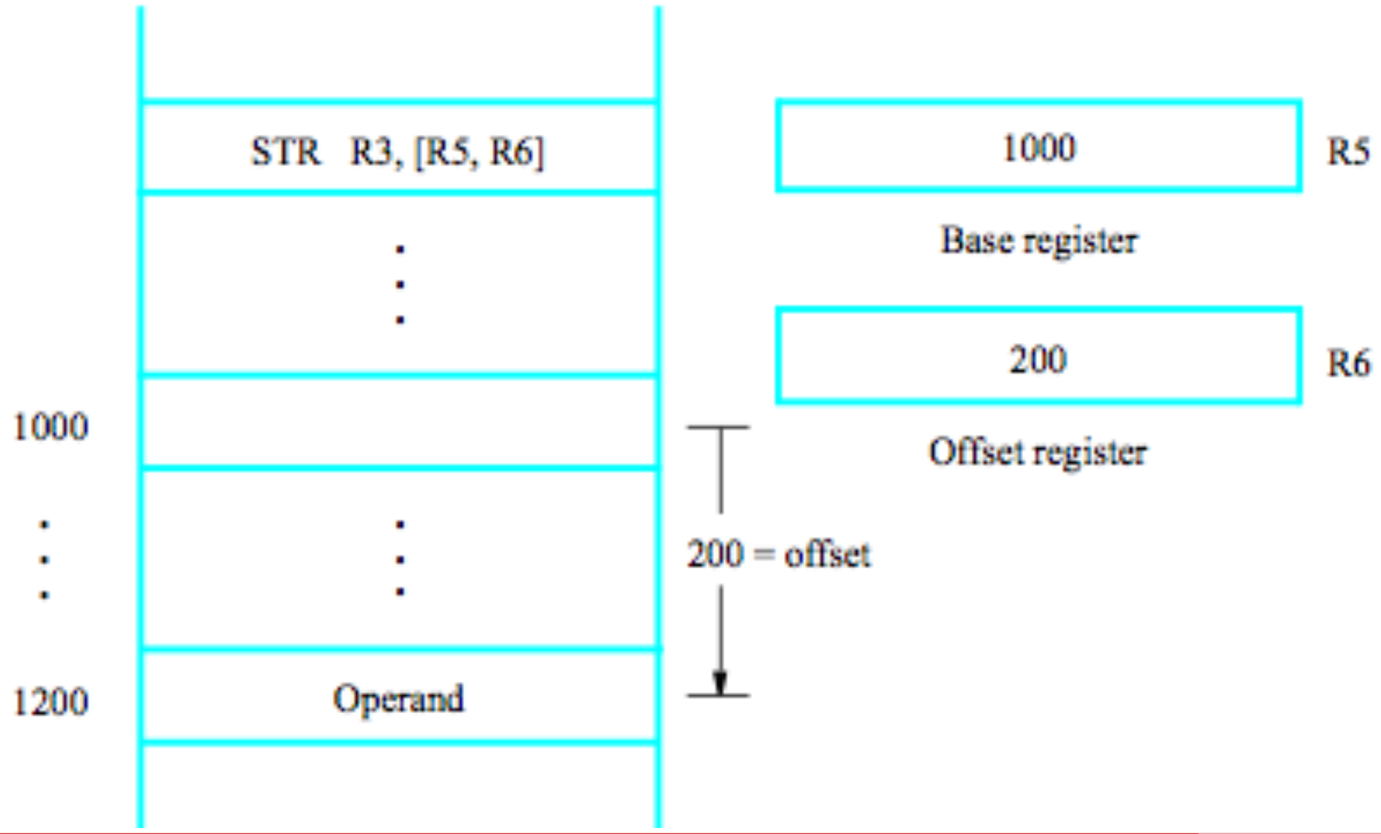
## Lecture Topics Today

- ISA
  - Examples
- Assembly Language and Tutorial Complement
  - Introduction to the ARM Cortex Assembler
  - Lab 1 Overview and Hints
  - Getting Started with STM32CubeIDE

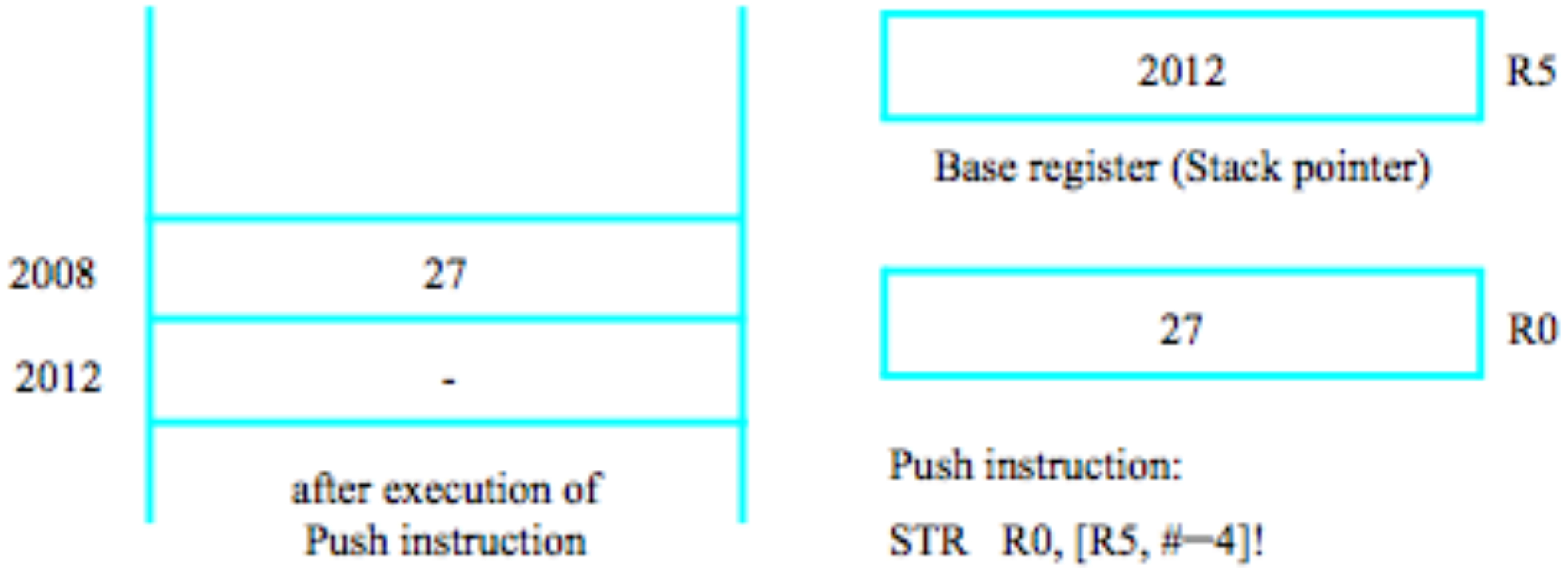
## Example: LDR Instruction Relative to PC



# Example: STR Instruction with Offset Addressing Mode



# Example: STR Instruction with Post-decrement Addressing


2008
27
2012
-
after execution of Push instruction
2012
R5
Base register (Stack pointer)
27
R0
Push instruction:
STR R0, [R5, #—4]!

## Arithmetic & Conditional Execution

- Mostly 3-operand (two inputs and output)

```
XYZ Rd, Rn, <Op2>
```

- Example:
  - ADD – add
  - ADC – add with carry
  - SUB – subtract, etc.

Examples:

| | |
|---|---|
| `ADD     R0, R1, R2` | R0=R1+R2, no flag change |
| `ADDS    R0, R1, R2` | as above, but CNZV affected |
| `ADDCSS  R0, R1, R2` | if C flag then ADDS |

## Branch Instructions

- Branch can be conditional or unconditional
- Syntax

```
B [cond] label
```

- Example
  - B – brach
  - BL – branch with link
  - BEQ – conditional branch

- Exampl

```
B        loopA  //      branch to loopA
BEQ      target //      conditional branch to target
BX       LR     //      return from function call
```

# Conditional Execution Example

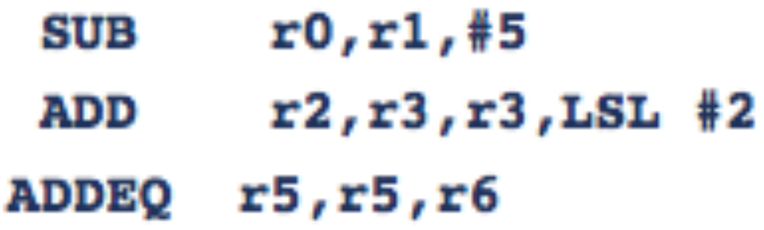
```
int gcd(int a, int b){
    while (a != b)
        if(a>b) a= a-b
            else b = b-a;
    return a;}
```

```
gcd:   CMP   R0, R1
       BEQ   end
       BLT   less
       SUB   R0, R0, R1
       B     gcd
less:  SUB   R1, R1, R0
       B     gcd
end:   ...
```

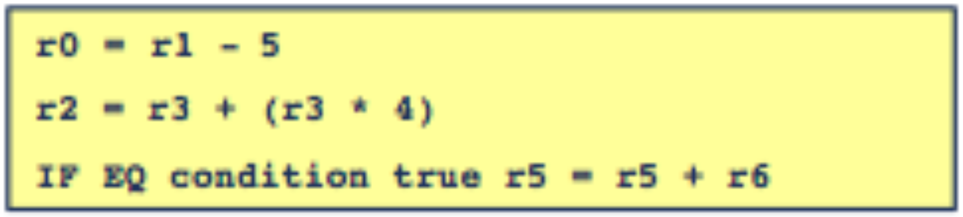
```
gcd:   CMP      R0, R1
       SUBGT    R0, R0, R1// SUB if R1 > R0
       SUBLT    R1, R1, R0// SUB if R1 < R0
       BNE      gcd
```

## ARM Instruction Set - Summary

- Original ARM instructions are 32 bit long, most execute in one cycle
- Instructions can be conditionally executed
- Example: data processing instruction



```
SUB     r0,r1,#5
ADD     r2,r3,r3,LSL #2
ADDEQ  r5,r5,r6
```

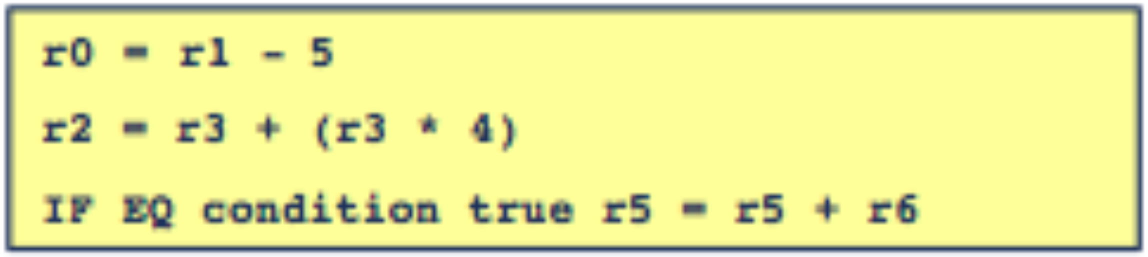


```
r0 = r1 - 5
r2 = r3 + (r3 * 4)
IF EQ condition true r5 = r5 + r6
```

## ARM Instruction Set - Summary

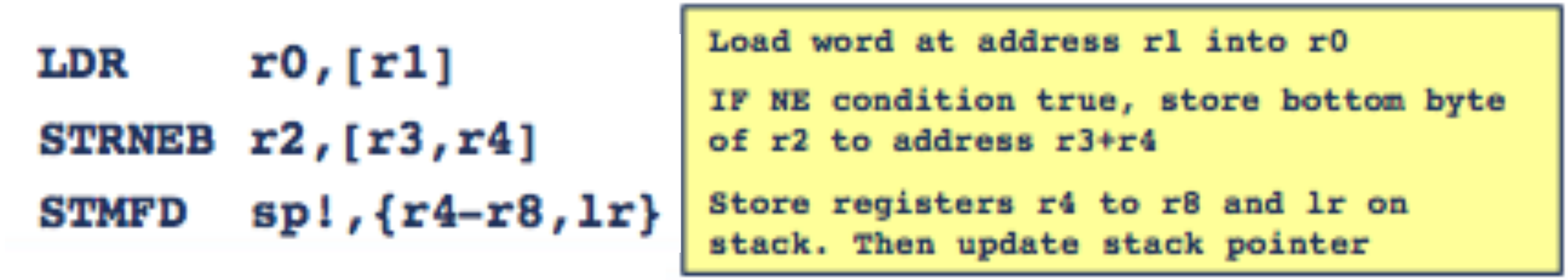
- Example: branching instruction

```
B    <Label>
```



```
r0 = r1 - 5
r2 = r3 + (r3 * 4)
IF EQ condition true r5 = r5 + r6
```

- Example: memory access instructions



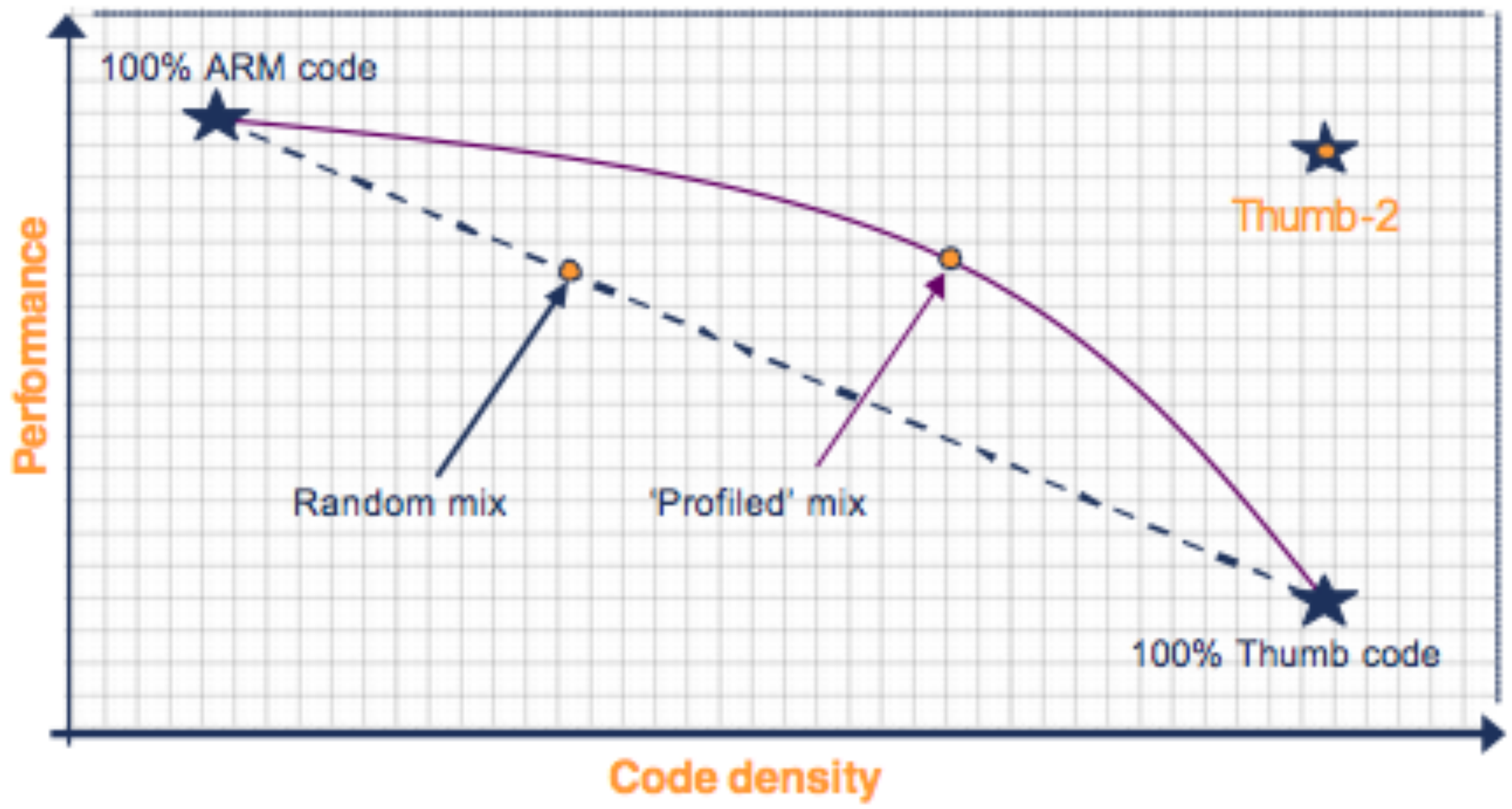
```
LDR     r0,[r1]
STRNEB  r2,[r3,r4]
STMFD   sp!,{r4-r8,lr}
```

```
Load word at address r1 into r0
IF NE condition true, store bottom byte
of r2 to address r3+r4
Store registers r4 to r8 and lr on
stack. Then update stack pointer
```

## Thumb Instruction Set

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of functionality of ARM instruction set
- Thumb is not a "regular" instruction set
  - Constraints are not generally consistent
  - Targeted at compiler generation, not hard coding

# Thumb Performance vs. Density



## Assembler

- Assembler is a utility program used to translate assembly language into machine code
  - Translation is nearly isomorphic (one-to-one) from assembly mnemonic statements into machine instructions and data
    - Note that the translation of a single high-level language instruction into machine code generally results in many machine instructions
    - In some cases assembler may provide pseudoinstructions expanding into several machine language instructions upon translation
      - Example: If assembly language is not supporting “branch if greater or equal” instruction then assembler can provide a pseudoinstrution expanding to “set if les than” and “branch id zero”

# Assembler - Pass 1

```
public static void pass_one( ) {                                           // This procedure is an outline of pass one of a simple assembler
boolean more_input = true;                                                 // flag that stops pass one
String line, symbol, literal, opcode;                                      // fields of the instruction
int location_counter, length, value, type;                                 // misc. variables
final int END_STATEMENT = -2;                                              // signals end of input
location_counter = 0;                                                      // assemble first instruction at 0
initialize_tables( );                                                      // general initialization
while (more_input) {                                                       // more_input set to false by END
    line = read_next_line( );                                              // get a line of input
    length = 0;                                                            // # bytes in the instruction
    type = 0;                                                              // which type (format) is the instruction
    if (line_is_not_comment(line)) { symbol = check_for_symbol(line);      // is this line labeled?
            if (symbol != null) enter_new_symbol(symbol, location_counter);   // if it is, record symbol and value
            literal = check_for_literal(line);                             // does line contain a literal?
            if (literal != null)  enter_new_literal(literal);              // if it does, enter it in table
            // Now determine the opcode type. -1 means illegal opcode.
            opcode = extract_opcode(line);                                 // locate opcode mnemonic
            type = search_opcode_table(opcode);                            // find format, e.g. OP REG1,REG2
            if (type < 0)  type = search_pseudo_table(opcode);             // if not an opcode, is it a pseudoinstruction?
            switch(type) {                                                 // determine the length of this instruction
                    case 1: length = get_length_of_type1(line); break;
                    case 2: length = get_length_of_type2(line); break;     // other cases here
            }
    }
write_temp_file(type, opcode, length, line);                               // useful info for pass two
location_counter = location_counter + length;// update loc_ctr
if (type == END_STATEMENT) {                                               // are we done with input?
    more_input = false;                                                    // if so, perform housekeeping tasks
    rewind_temp_for_pass_two( );                                           // like rewinding the temp file
    sort_literal_table( );                                                 // and sorting the literal table
    remove_redundant_literals( );                                          // and removing duplicates from it
    }
}
```

# Assembler - Pass 2

```
public static void pass_two( )                          // This is an outline of pass two of a simple assembler
boolean more_input = true;                              // flag that stops pass one of assembler
String line, opcode;                                    // fields of the instruction
int location_counter, length, type;                     // misc. variables
final int END_STATEMENT = -2;                           // signals end of input
final int MAX_CODE = 16;                                // max bytes of code per instruction
byte code[ ] = new byte[MAX_CODE];                      // holds generated code per instruction
location_counter = 0;                                   // assemble first instruction at 0
while (more_input) {                                    // more_input set to false by END
        type = read_type( );                            // get type field of next line
        opcode = read_opcode( );                        // get opcode field of next line
        length = read_length( );                        // get length field of next line
        line = read_line( );                            // get the actual line of input
        if (type != 0) {                                 // type 0 is for comment lines
            switch(type) {                              // generate the output code
                     case 1: eval_type1(opcode, length, line, code); break;
                     case 2: eval_type2(opcode, length, line, code); break;
                                                        // other cases here
                     }
        }
write_output(code);                                     // write the binary code
write_listing(code, line);                              // print one line on the listing
location_counter = location_counter + length;           // update loc_ctr
if (type == END_STATEMENT) {                            // are we done with input?
        more_input = false;                             // if so, perform housekeeping tasks
        finish_up( );                                   // odds and ends
        }
}
```

# ARM Assembly Language

- Assemblers: nearly as powerful as high-level languages
  - ARM Assembler "feels" somewhat differently
- One (machine) command per line, but with ARM, could expand
- Assembly code: commands or directives (which are commands to assembler tool, rather than processor)

Format:

```
{label:}{instruction | directive | pseudo-instruction}//
    comment}
```

Notes:

- Everything is optional
- Label must start  the line (no spaces or tabs before)
- Spaces and tabs freely used otherwise
- Comments can't spread multiple lines

# Assembly Directives: GNU & ARM

```
          .section .text,"x”                       //AREA  PROGRAM, CODE
//subroutine that takes 2 7-digit BCD numbers and places results in the first one
// bcdadd(a, b)-> a' with a'=a+b for encoding below
//  svxxD6 D5D4 D3D2 D1D0; s=1 is negative, v=1 is overflow, 7  4-bit BCD Di
//ARM Calling convention: arg1 and arg2 in R0 and R1, respectively. Result in R0
//other registers unchanged. Need to be pushed to stack if changed, and pop-ed
before return
// no memory used
//return via LR register
//position-independent code, needs to be linked with C routine
          .global bcdadd                           //EXPORT bcdadd
          .align 4                                 //ENTRY
bcdadd:   // label of the subroutine entry point, the rest of code comes after
...
          .end                                     //END
```

## Example: Assembly Directives,

- ..segment (AREA) directive instructs the assembler to assembly a new code or data section
- Syntax

```
    .segment { , attr}{,attr}…
```

- .text (CODE) is a keyword indicating that the section to follow is a code
- .global (EXPORT) is a keyword indicating functions, variables, etc. visible to external segment

# Assembler Directives - GNU

- Important directives are these for data definition and storage
  - Equivalent of high-level language
    - Symbolic constant (to be replaced throughout code): .equ
      - .equ directive gives symbolic name to a numeric constant, a register-relative value or a PC-relative value
    - Variable: .word .float .byte .ascii etc.
      - .word (.byte) directive allocates one or more words (bytes) of memory. Words are aligned on four-byte boundaries
      - .float directive allocated one or more floating-point numbers and defines the initial runtime contents of the memory

- Examples:
  - Array: .word 1, 2, 3 //array, not only 1 word
  - FailingGrade: .ascii “D, C-, C, or C+” //string
  - .equ PerfectNumber, 10 // Pythagora ~500 B.C
  - .equ LF, 10 // linefeed in ASCII

# Assembler Directives - ARM

- Important directives are these for data definition and storage
  - Equivalent of high-level language
    - Symbolic constant (to be replaced throughout code): EQU
      - EQU directive gives symbolic name to a numeric constant, a register-relative value or a PC-relative value
    - Variable: DCD, DCB
      - DCD directive allocates one or more words of memory, aligned on four-byte boundaries
      - DCB directive allocated one or more bytes of memory, and defines the initial runtime contents of the memory
    - Register variable: RN
    - Debug support: INFO, ASSERT
- Examples:

| | |
|---|---|
| Array DCD 1, 2, 3 | ;array, not only 1 word |
| FailingGrade DCB “D, C-, C, or C+”, 0 | |
| INFO 0, “Version 1.0” | ;reserves 10 bytes |
| PerfectNumber EQU 10 | ; Pythagora ~500 B.C |
| LF EQU 10 | ; linefeed in ASCII |
| Area  RN R5 | ; R5 holds var Area |

# Cortex ISA and Assembler: Hints

- Note that some ARM instructions dropped

Example: no LDR Rt, [Rn, Rm, LSL #imm] but LDR Rt, [Rn, #imm]! is OK

- no writeback with shift[ARMV7-M App. Level Ref. Manual, p. A7-291]

- Use conditional execution -> avoid branches
  - Simpler code
  - Full pipeline
- Useful instructions for bit-field operations:

BIC, BFC - sets some bits to zero

BFI - inserts bit fields to a word

STMDB, LDMIA - push & pop (decrement before, increment after)

AND, ORR, EOR - logical bit manipulations

- Rely on STM tool and documentation posted on Web
  - Instruction Set Quick Reference Card