ECSE 426 – Fall 2015
Microprocessor Systems

Dept. Electrical and Computer Engineering
McGill University

Hidden Markov Models

- We'll discuss HMMs and the Viterbi algorithm in class

Viterbi algorithm

```
def viterbi (self,observations):
    """Return the best path, given an HMM model + sequence of observations"""
    # A - initialisation
    nSamples = len(observations[0])
    nStates = self.transition.shape[0] # number of states
    c = np.zeros(nSamples) # scale factors (necessary to prevent underflow)
    vit = np.zeros((nStates,nSamples)) # initialise viterbi table
    psi = np.zeros((nStates,nSamples)) # initialise the best path table
    vit_path = np.zeros(nSamples); # initialize best sequence

    # B - insert initial values into viterbi and best path (bp) tables
    vit[:,0] = self.priors.T * self.emission[:,observations(0)]
    c[0] = 1.0/np.sum(viterbi[:,0])
    vit[:,0] = c[0] * viterbi[:,0] # apply the scaling factor
    psi[0] = 0;
```

Viterbi algorithm

```
# C- Viterbi iterations for time>0 until T
for t in range(1,nSamples): # loop through time
   for s in range (0,nStates): # loop through the states
      trans_p = vit[:,t-1] * self.transition[:,s]
      psi[s,t], vit[s,t] = max(enumerate(trans_p), key=operator.itemgetter(1))
      vit[s,t] = vit[s,t]*self.emission[s,observations(t)]

   c[t] = 1.0/np.sum(vit[:,t]) # scaling factor
   vit[:,t] = c[t] * vit[:,t]

# D - Back-tracking
vit_path[nSamples-1] =  vit[:,nSamples-1].argmax() # last state
for t in range(nSamples-1,0,-1): # states of (last-1)th to 0th time step
   vit_path[t-1] = psi[vit_path[t],t]
```

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

Assembler

- Utility program: translate assembly language into machine code
  - Translation is nearly isomorphic (one-to-one)
  - Assembly mnemonic statements ⇨ machine instructions and data

  - Translation of a single high-level language instruction into machine code generally results in many machine instructions
  - Assembler may provide pseudo-instructions
    - expand into several machine language instructions
    - Example:
      - No support for "branch if greater or equal" instruction
      - Assembler can provide pseudo-instruction that expands to "set if less than" and "branch if zero"

ARM Assembly Language

- Assemblers: nearly as powerful as high-level languages

- Assembly code: commands or directives (commands to the assembler tool rather than the 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 over multiple lines

Example: Assembly Directives, cont.

- AREA directive instructs assembler to assemble a new code/data section
- Section: independent, named, indivisible code chunk (manipulated by linker)
- Syntax

```
AREA sectionname { , attr}{,attr}…
```

  - sectionname: name given to the section. Any can be chosen

- CODE: keyword indicating that the section to follow is code
- EXPORT: indicates functions, variables, etc. are visible to external segment
- ENTRY: keyword indicating the entry point of a segment

Example: Assembly Directives

```
        AREA   PROGRAM, CODE
; takes 2 7-digit BCD numbers and places results in the first
; bcdadd(a, b)-> a' with a'= a+b for encoding below
; svxxD6 D5D4 D3D2 D1D0 where s=1 is negative, v=1 is
; overflow, and 7 4-bit BCD Di
; ARM Calling convention: arg1 and arg2 in R0 and R1,
; respectively. Result is in R0 & other registers unchanged.
; no memory used; return via LR register
; position-independent code, needs to be linked with C routine
        EXPORT bcdadd
        ENTRY
bcdadd ; labels entry point to subroutine & rest of code after
…
        END
```

Assembler Directives

- Important directives for data definition and storage
  - Symbolic constant (to be replaced throughout code): EQU
    - EQU directive gives symbolic name to:
      - numeric constant, a register-relative value or a PC-relative value
  - Variable: DCD, DCB
      - allocate one or more words of memory
      - defines the initial runtime contents of the memory
      - DCD: aligned on four-byte boundaries
  - Register variable: RN (define a name for a specified register)
  - Debug support: INFO, ASSERT
    - INFO supports diagnostic generation
    - ASSERT generates error message during assembly if assertion is false

Assembler Directives

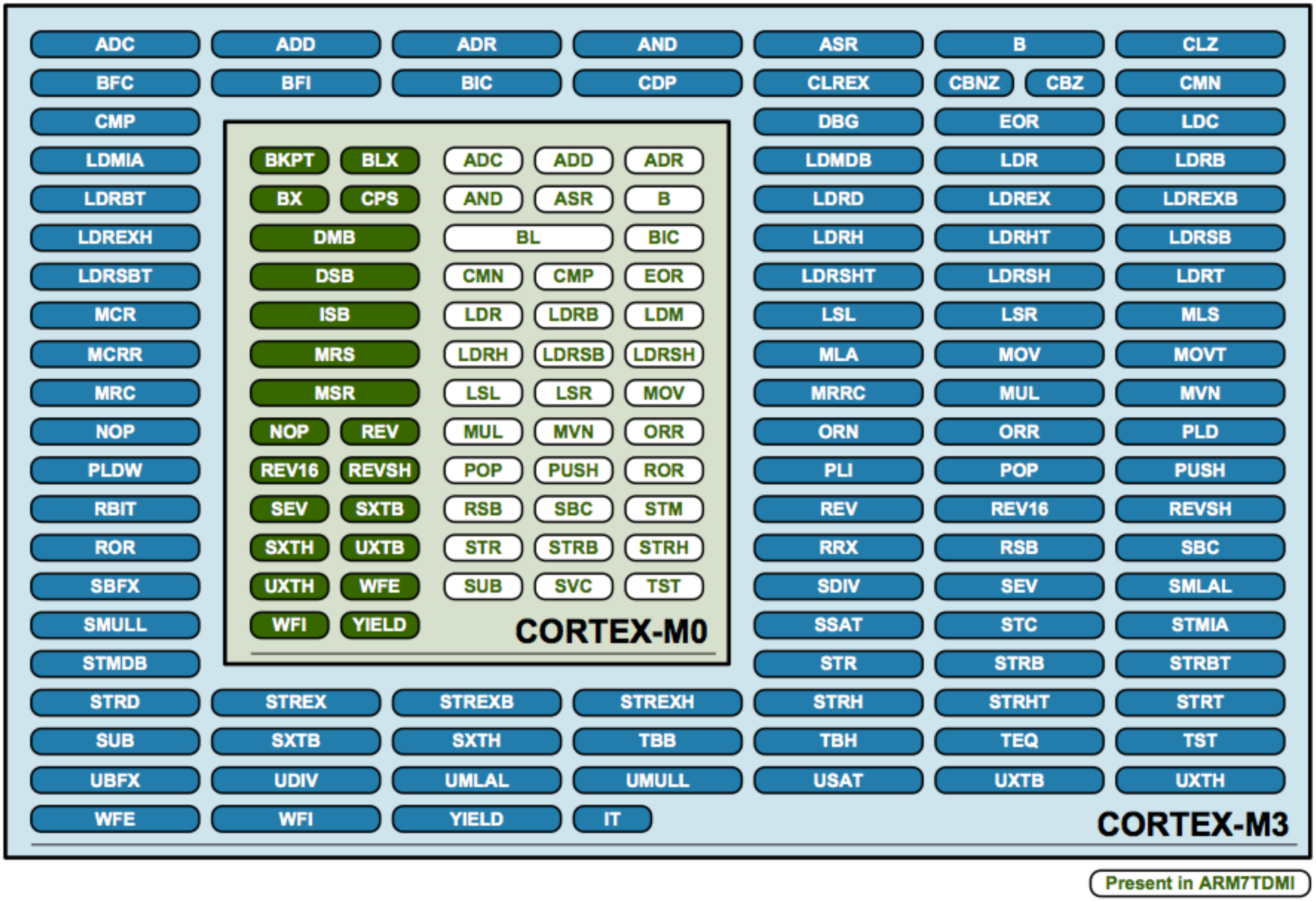
- Examples:
  - Array DCD 1, 2, 3 ; array, not only 1 word
  - FailingGrade DCB “D, C-, C, or C+”, 0
  - INFO 0, “Version 1.0” ;reserves 10 bytes
  - PerfectNumber EQU 10 ; Pythagora ~500 B.C
  - LF EQU 10 ; linefeed in ASCII
  - Area RN R5 ; R5 holds var Area

Cortex ISA and Assembler: Hints

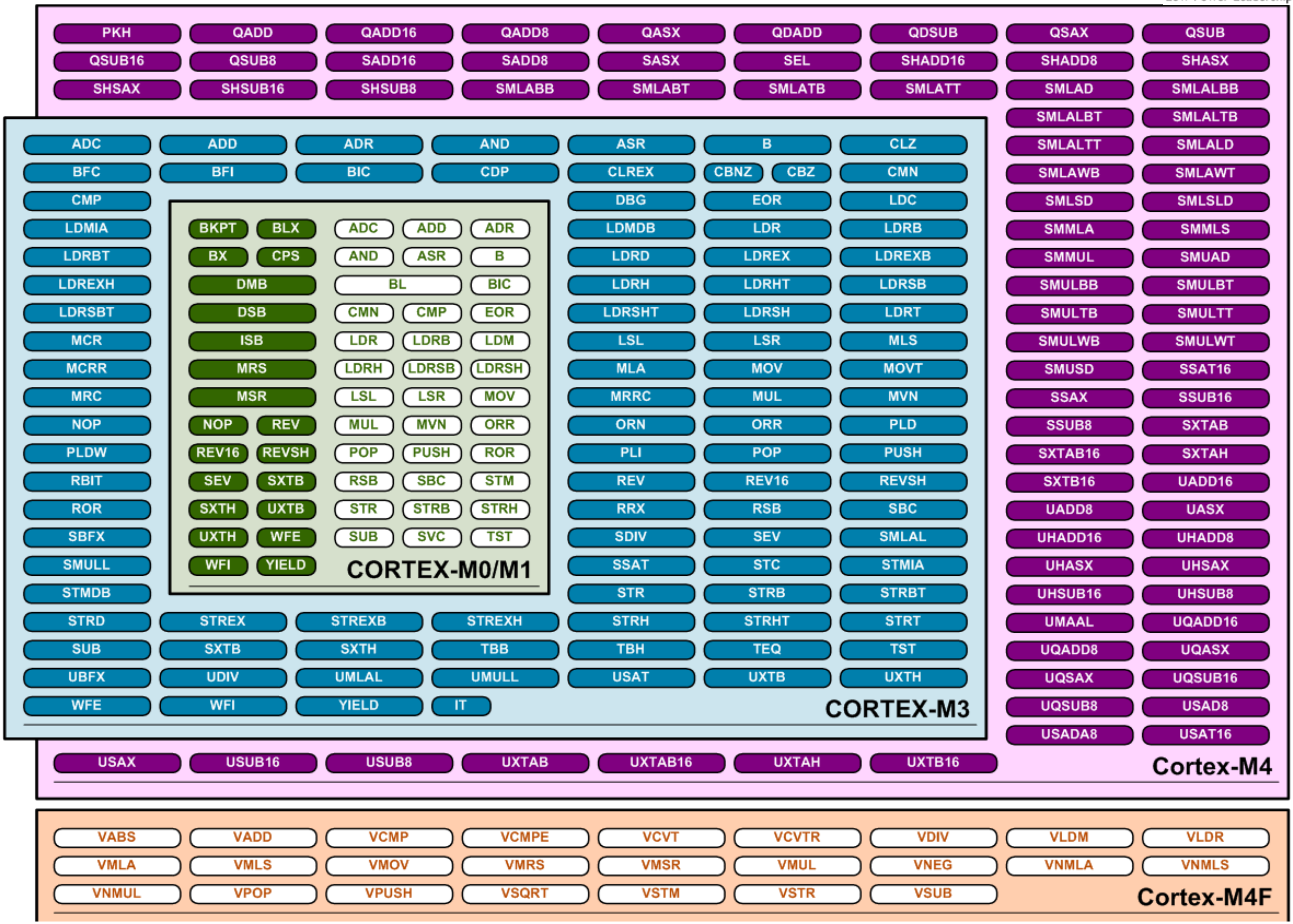
- Use conditional execution -> avoid branches
  - Simpler code
  - Full pipeline

- Useful instructions for bit-field operations:
  - BIC, BFC - sets some bits to zero
  - BFI - inserts bit fields to a word
  - AND, ORR, EOR - logical bit manipulations

Cortex M3 ISA at Glance



Cortex-M4 processors



Embedded C

- C preprocessor
  - #define, #ifndef, #if, #ifdef, #else …etc.
    - #define specifies flags for conditional compilation
    - All remaining preprocessor statements initiate conditional compilation
      - Example: #ifdef compiles a block of code if some condition is defined in the #define statement
  - #ifndef is used to prevent multiple includes
    - Use #ifndef if you want to include a new definition
  - Widely used in firmware (embedded SW) drivers

C Preprocessor Examples

Inline macro functions:

```
#define MIN(n,m) (((n) < (m)) ? (n) : (m))

#define MAX(n,m) (((n) < (m)) ? (m) : (n))

#define ABS(n) ((n < 0) ? -(n) : (n))
```

Macro used to set LCD control
(## is used to actual arguments during macro expansion)

```
#define SET_VAL(x) LCD_Settings.P##x
```

Nested macro definitions

```
#define SET(x, val) SET_VAL(x) = val
#define DEF_SET(x) SET(x, DS_P##x)
```

Global variables

- Distinguish global variables from local by choosing appropriate naming convention
  - Example: RX_Buffer_Gbl
  - Stick to your convention throughout the program

- Use them as Software flags
  - Example: PACKET_RECEIVED – use capitals

- Have them all in ONE place

- Global variables are easy to observe during debug (“watch variables”)