CprE 488 – Embedded Systems Design

Lecture 3 – Processors and Memory

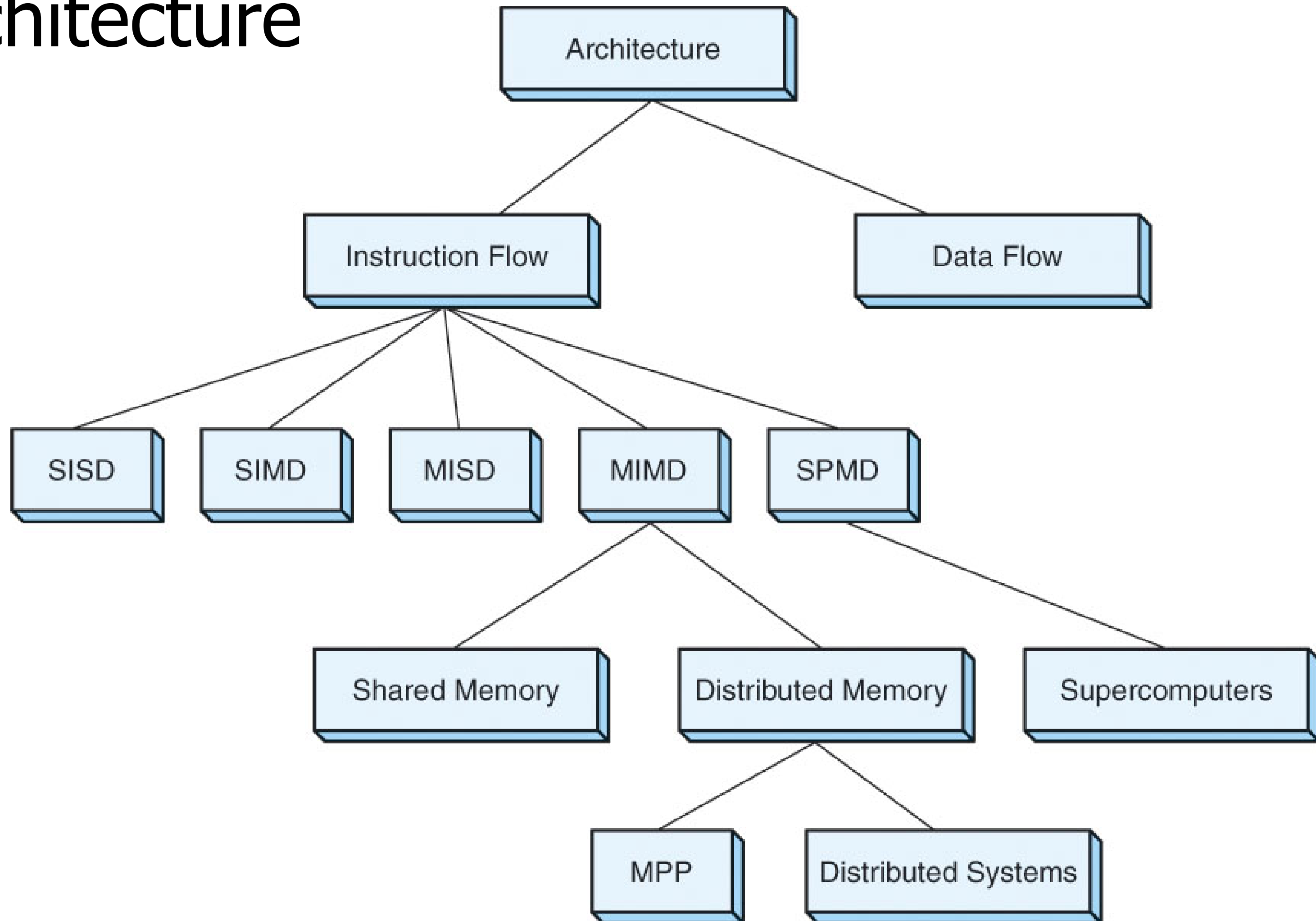
Joseph Zambreno
Electrical and Computer Engineering
Iowa State University

www.ece.iastate.edu/~zambreno
rcl.ece.iastate.edu

Although computer memory is no longer expensive, there's always a finite size buffer somewhere. – Benoit Mandelbrot

Flynn's (Updated) Taxonomy

- AKA the "alphabet soup" of computer architecture

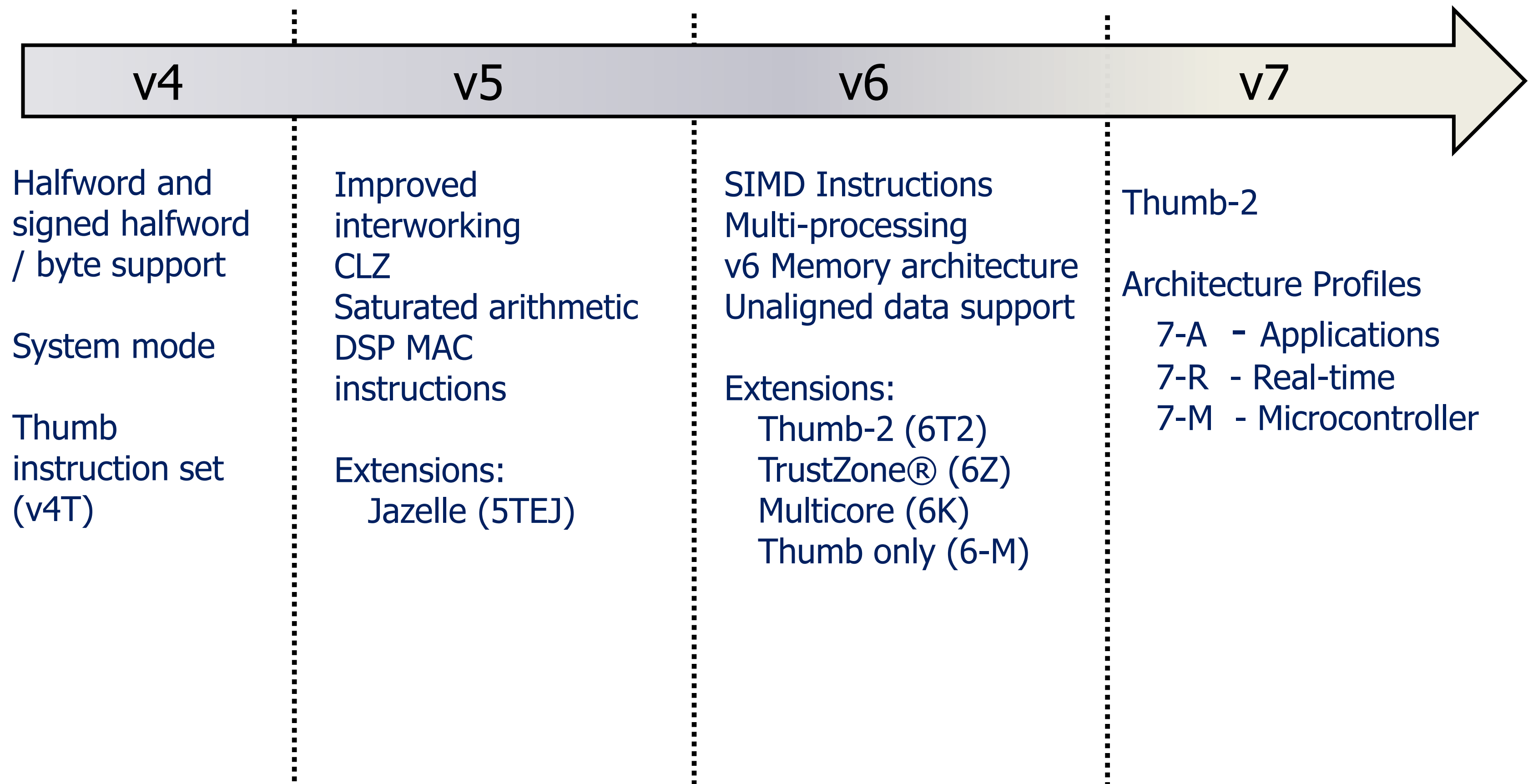


This Week's Topic

- Embedded processor design tradeoffs
- ISA and programming models
- Memory system mechanics
- Case studies:
  - ARM v7 CPUs (RISC)
  - TI C55x DSPs (CISC)
  - TI C64C (VLIW)

- Reading: Wolf chapters 2, 3.5

ARM Architecture Revisions



- Note that implementations of the same architecture can be different
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

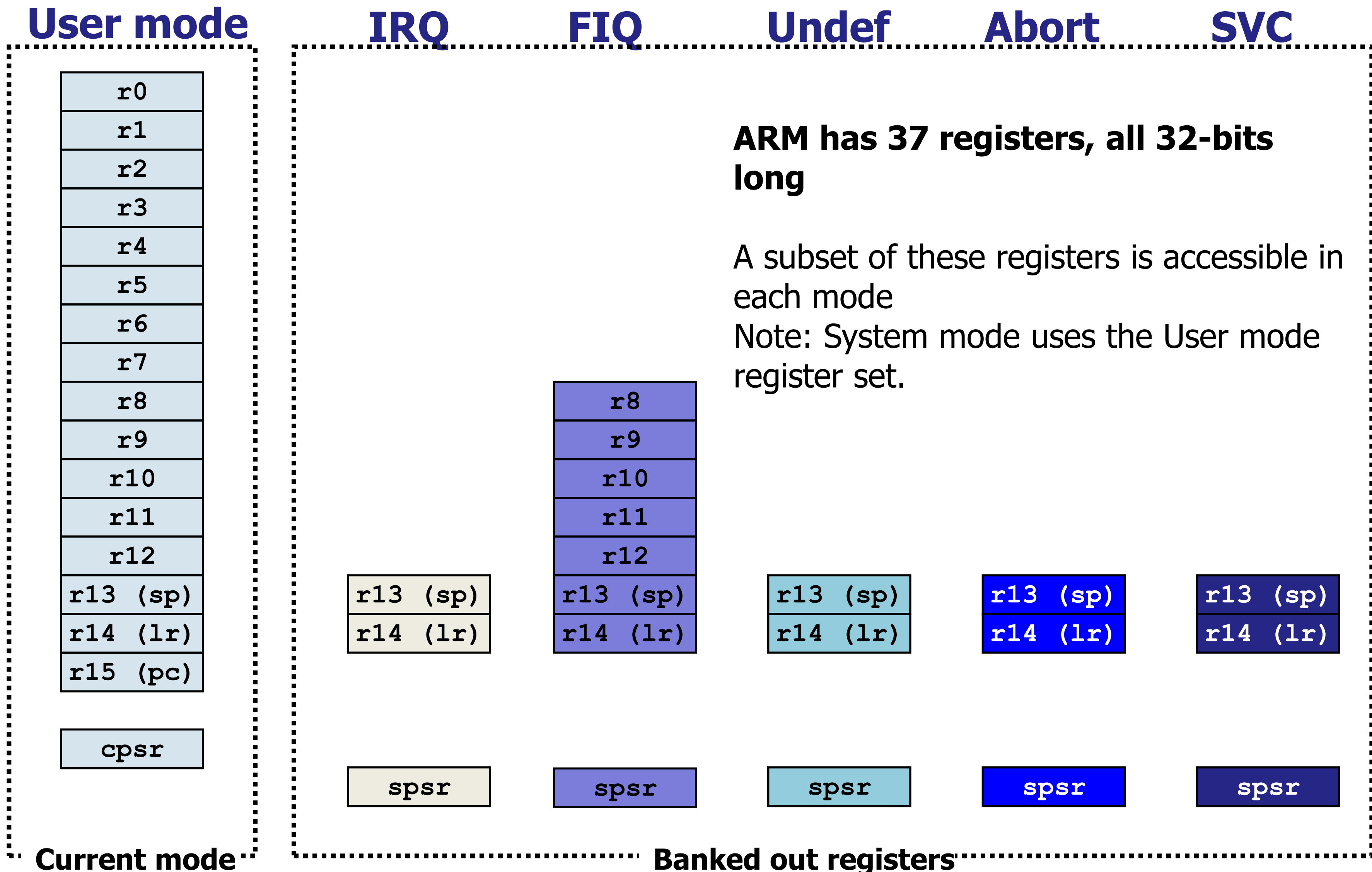
Data Sizes and Instruction Sets

- ARM is a 32-bit load / store RISC architecture
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide

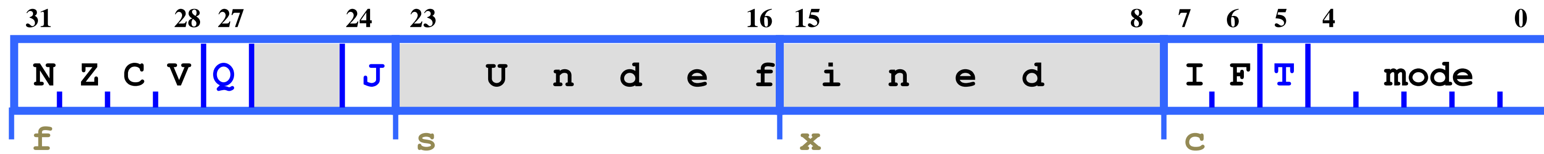
- ARM cores implement two basic instruction sets
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set

- Depending on the core, may also implement other instruction sets
  - **VFP** instruction set – 32 bit (vector) floating point instructions
  - **NEON** instruction set – 32 bit SIMD instructions
  - **Jazelle-DBX** - provides acceleration for Java VMs (with additional software support)
  - **Jazelle-RCT** - provides support for interpreted languages

The ARM Register Set



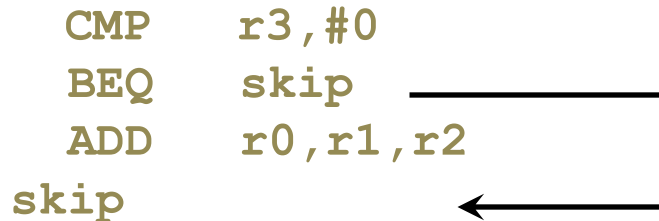
Program Status Registers



- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state
- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.
- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- Mode bits
  - Specify the processor mode

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

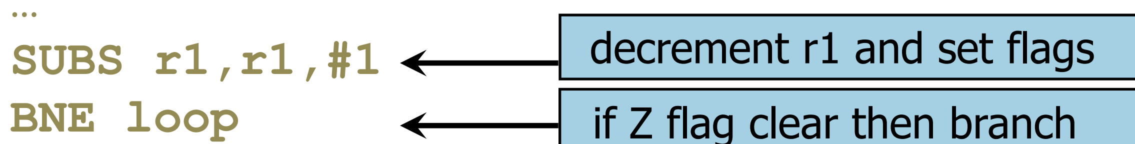


```
    CMP     r3,#0
    BEQ     skip
    ADD     r0,r1,r2
skip
```

```
CMP     r3,#0
ADDNE   r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
    …
    SUBS r1,r1,#1
    BNE  loop
```



Condition Codes

- The possible condition codes are listed below
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

Conditional Execution Examples

C source code

```
if (r0 == 0)
{
  r1 = r1 + 1;
}
else
{
  r2 = r2 + 1;
}
```

ARM instructions

unconditional

```
  CMP r0, #0
  BNE else
  ADD r1, r1, #1
  B end
else
  ADD r2, r2, #1
end
  ...
```

- 5 instructions
- 5 words
- 5 or 6 cycles

conditional

```
  CMP r0, #0
  ADDEQ r1, r1, #1
  ADDNE r2, r2, #1
  ...
```

- 3 instructions
- 3 words
- 3 cycles

Data Processing Instructions

- Consist of :
  - Arithmetic: `ADD ADC SUB SBC RSB RSC`
  - Logical: `AND ORR EOR BIC`
  - Comparisons: `CMP CMN TST TEQ`
  - Data movement: `MOV MVN`

- These instructions only work on registers, NOT memory

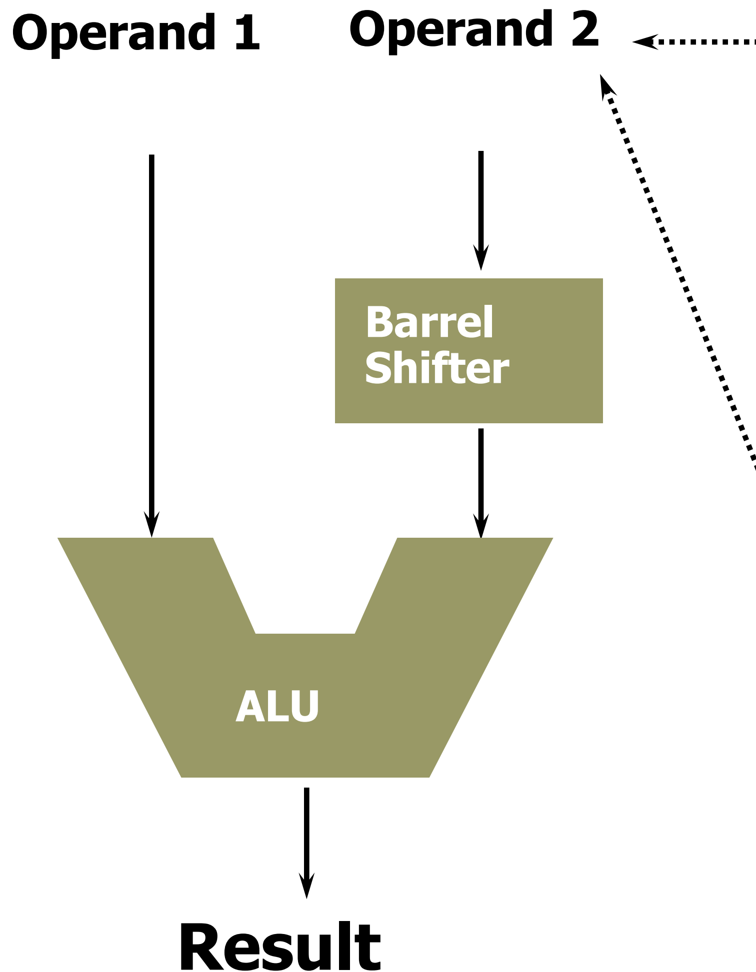
- Syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.

Using a Barrel Shifter



Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

Data Processing Exercise

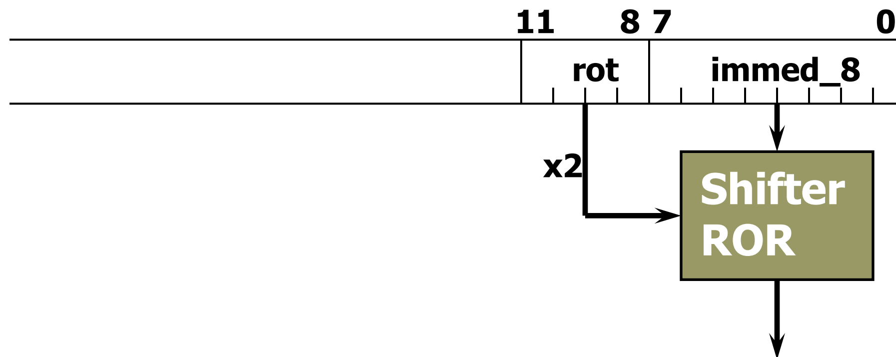
1. How would you load the two's complement representation of -1 into Register 3 using one instruction?

2. Implement an ABS (absolute value) function for a registered value using only two instructions

3. Multiply a number by 35, guaranteeing that it executes in 2 core clock cycles

Immediate Constants

- No ARM instruction can contain a 32 bit immediate constant
  - All ARM instructions are fixed as 32 bits long
- The data processing instruction format has 12 bits available for operand2



Quick Quiz:

```
0xe3a004ff
MOV r0, #???
```

- 4 bit rotate value (0-15) is multiplied by two to give range 0-30 in steps of 2
- Rule to remember is

  "8-bits rotated right by an even number of bit positions"

Single Register Data Transfer

LDR	**STR**	Word
LDRB	**STRB**	Byte
LDRH	**STRH**	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Memory system must support all access sizes
- Syntax:
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>

  e.g. **LDREQB**

Address Accessed

- Address accessed by LDR/STR is specified by a base register with an offset
- For word and unsigned byte accesses, offset can be:
  - An unsigned 12-bit immediate value (i.e. 0 - 4095 bytes)

    ```
    LDR r0, [r1, #8]
    ```

  - A register, optionally shifted by an immediate value

    ```
    LDR r0, [r1, r2]
    LDR r0, [r1, r2, LSL#2]
    ```

- This can be either added or subtracted from the base register:

  ```
  LDR r0, [r1, #-8]
  LDR r0, [r1, -r2, LSL#2]
  ```

- For halfword and signed halfword / byte, offset can be:
  - An unsigned 8 bit immediate value (i.e. 0 - 255 bytes)
  - A register (unshifted)
- Choice of *pre-indexed* or *post-indexed* addressing
- Choice of whether to update the base pointer (pre-indexed only)

  ```
  LDR r0, [r1, #-8]!
  ```

Load/Store Exercise

Assume an array of 25 words. A compiler associates y with r1. Assume that the base address for the array is located in r2. Translate this C statement/assignment using just three instructions:

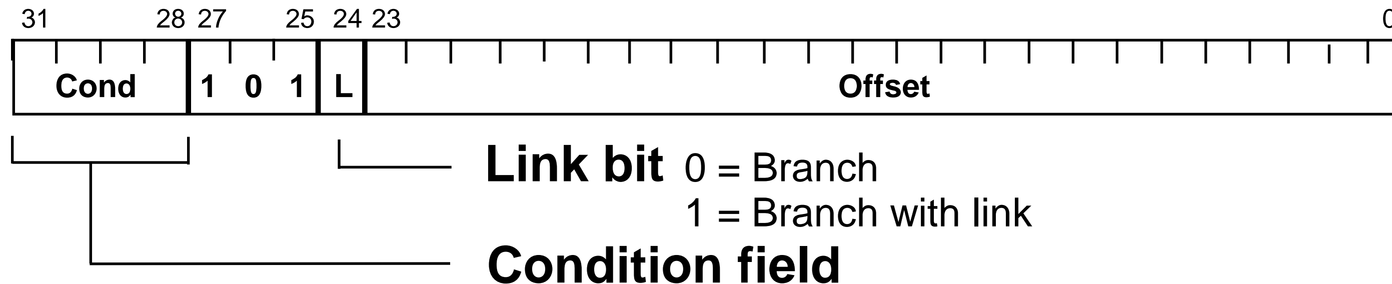
```
array[10] = array[5] + y;
```

Multiply and Divide

- There are 2 classes of multiply - producing 32-bit and 64-bit results
- 32-bit versions on an ARM7TDMI will execute in 2 - 5 cycles
  - `MUL r0, r1, r2                ; r0 = r1 * r2`
  - `MLA r0, r1, r2, r3            ; r0 = (r1 * r2) + r3`
- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers
  - `[U|S]MULL r4, r5, r2, r3      ; r5:r4 = r2 * r3`
  - `[U|S]MLAL r4, r5, r2, r3      ; r5:r4 = (r2 * r3) + r5:r4`
- Most ARM cores do not offer integer divide instructions
  - Division operations will be performed by C library routines or inline shifts

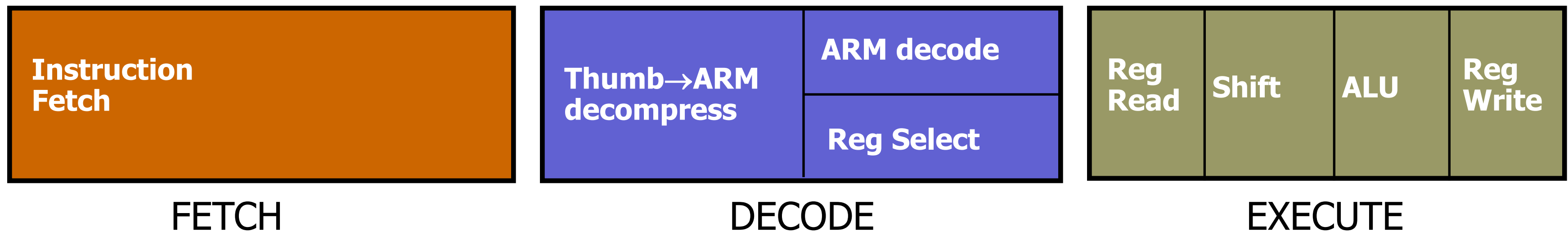
Branch Instructions

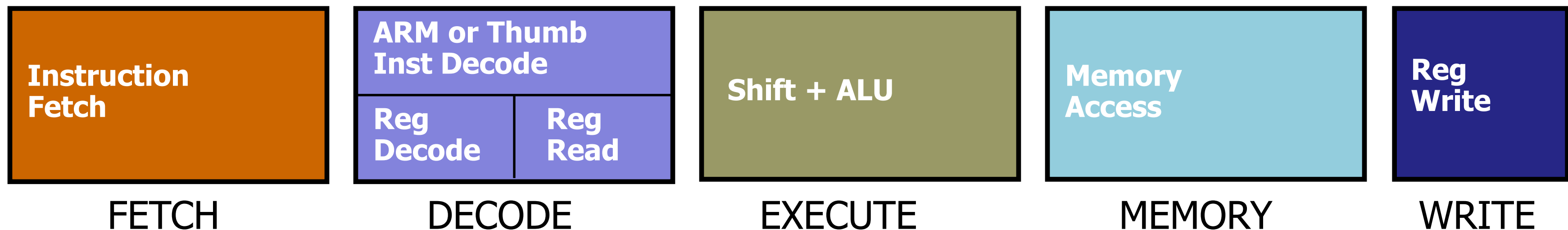
- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



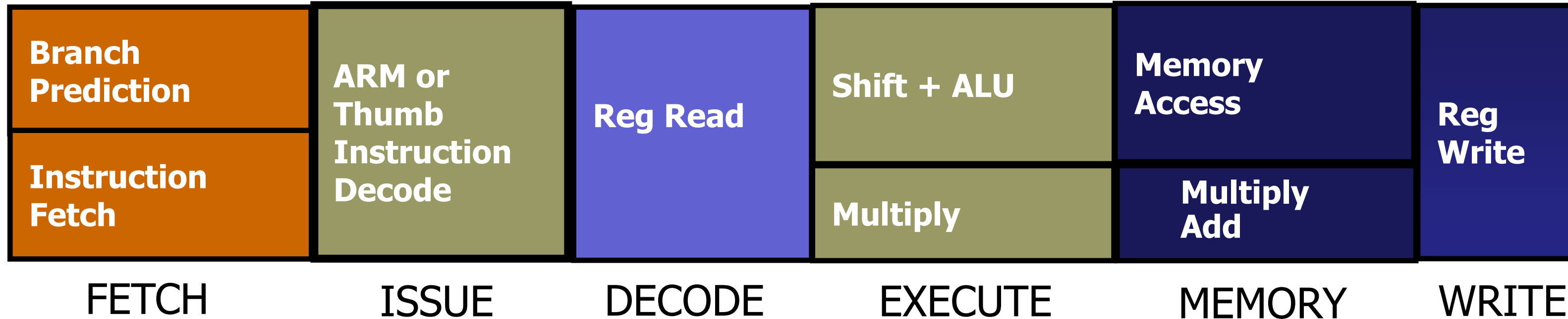
- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

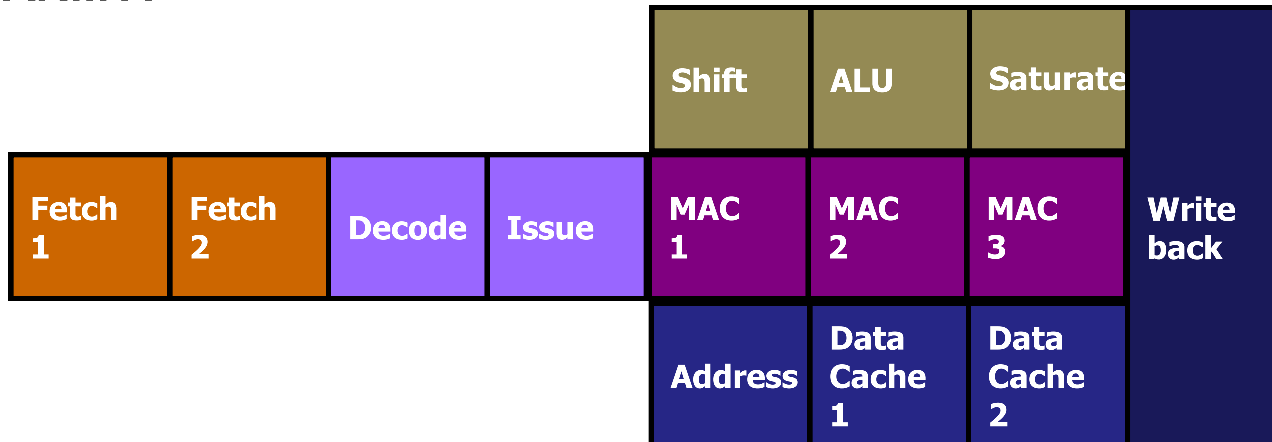
ARM Pipeline Evolution



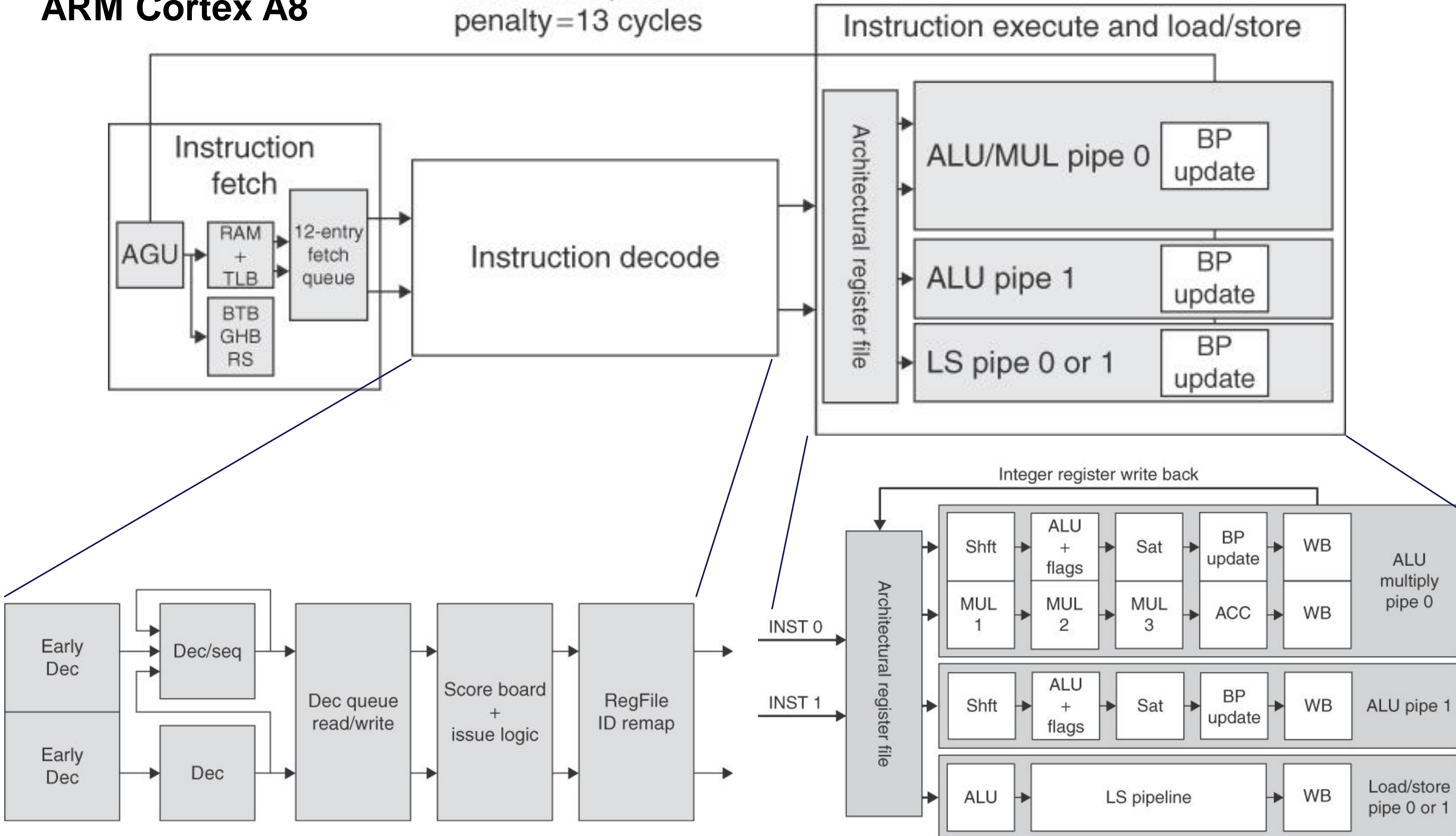


ARM Pipeline Evolution (cont.)



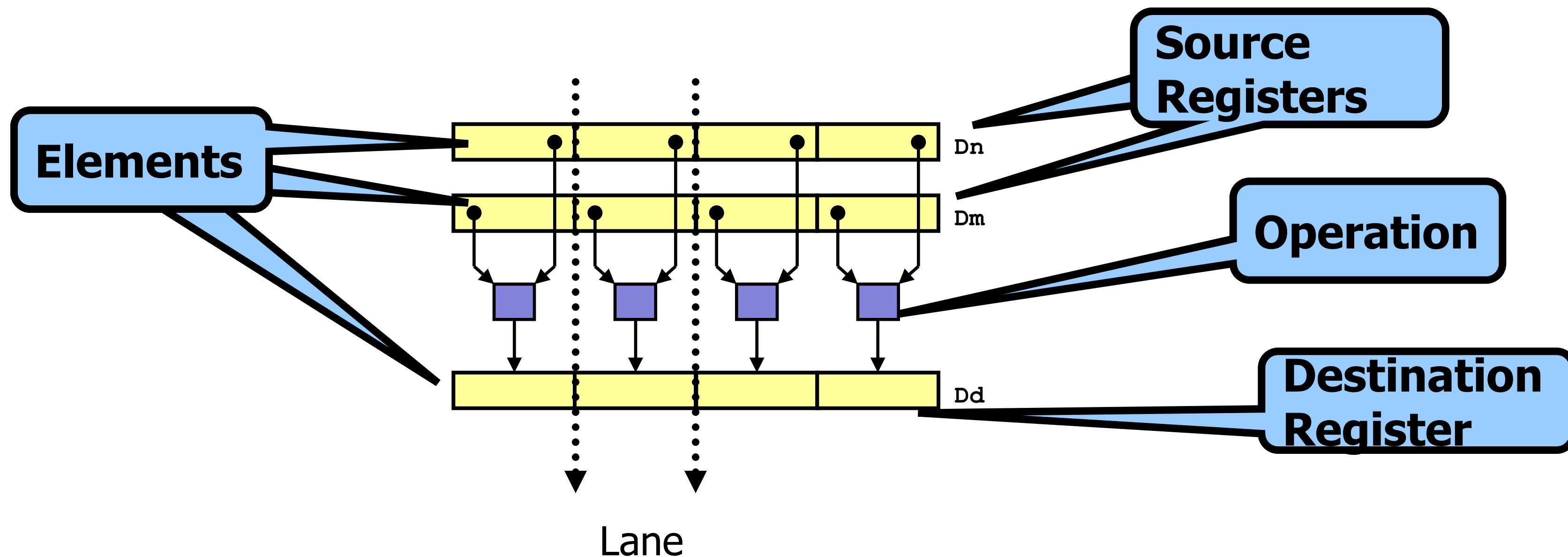


ARM Pipeline Evolution (cont.)



What is NEON?

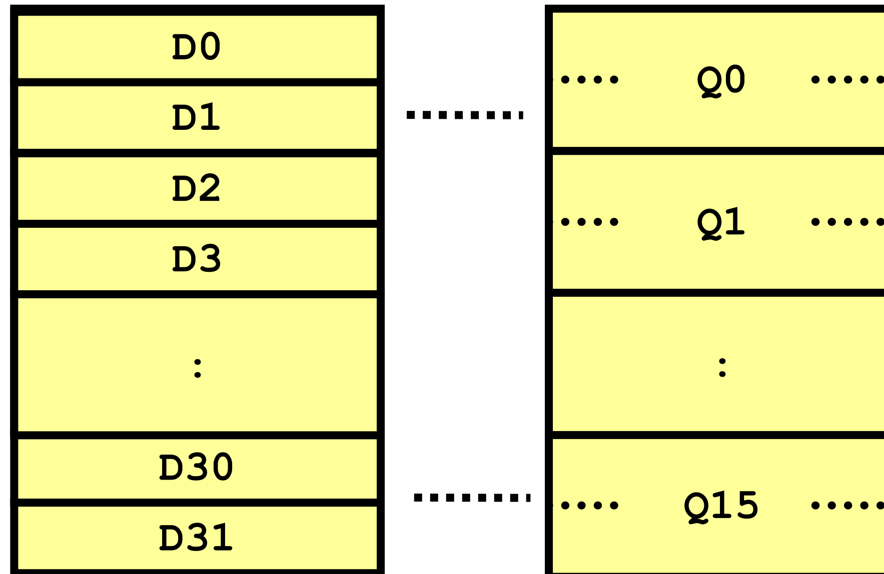
- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set (v7-A)
  - 32 x 64-bit wide registers (can also be used as 16 x 128-bit wide registers)



- NEON instructions perform "Packed SIMD" processing
  - Registers are considered as **vectors** of **elements** of the same data type
  - Data types available: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions usually perform the same operation in all **lanes**

NEON Coprocessor Registers

- NEON has a 256-byte register file
  - Separate from the core registers (r0-r15)
  - Extension to the VFPv2 register file (VFPv3)
- Two different views of the NEON registers
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)



- Enables register trade-offs
  - Vector length can be variable
  - Different registers available

NEON Vectorizing Example

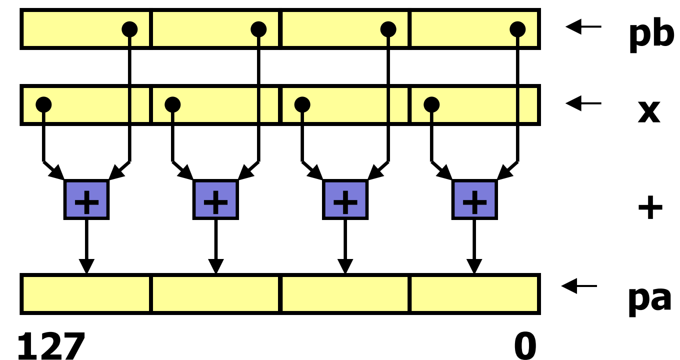
- How does the compiler perform vectorization?

```
void add_int(int * __restrict pa,
             int * __restrict pb,
             unsigned int n, int x)
{
  unsigned int i;
  for(i = 0; i < (n & ~3); i++)
    pa[i] = pb[i] + x;
}
```

1. Analyze each loop:
   - Are pointer accesses safe for vectorization?
   - What data types are being used? How do they map onto NEON vector registers?
   - Number of loop iterations

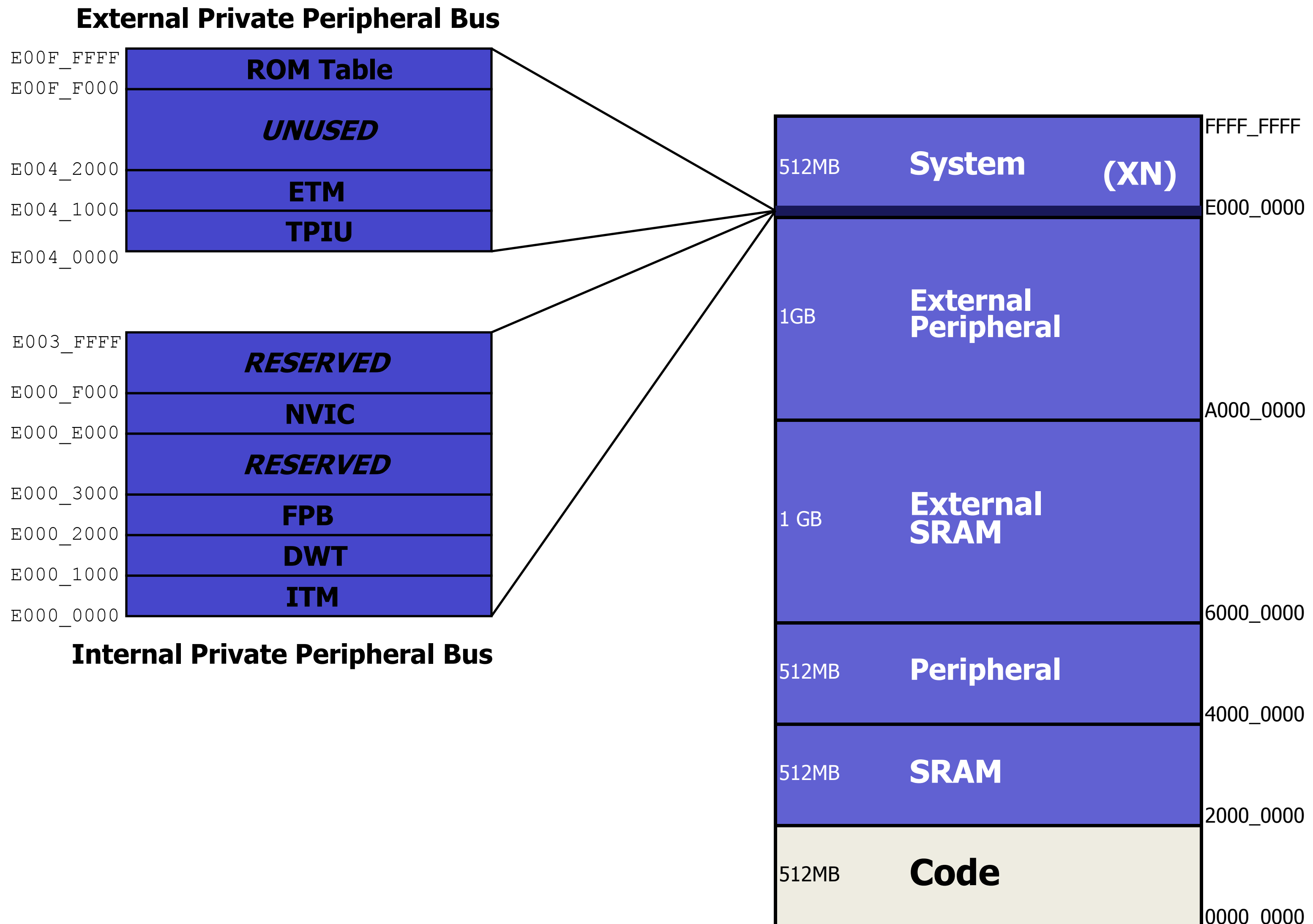
2. Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization

```
void add_int(int *pa, int *pb,
             unsigned n, int x)
{
  unsigned int i;
  for (i = ((n & ~3) >> 2); i; i--)
  {
    *(pa + 0) = *(pb + 0) + x;
    *(pa + 1) = *(pb + 1) + x;
    *(pa + 2) = *(pb + 2) + x;
    *(pa + 3) = *(pb + 3) + x;
    pa += 4; pb += 4;
  }
}
```

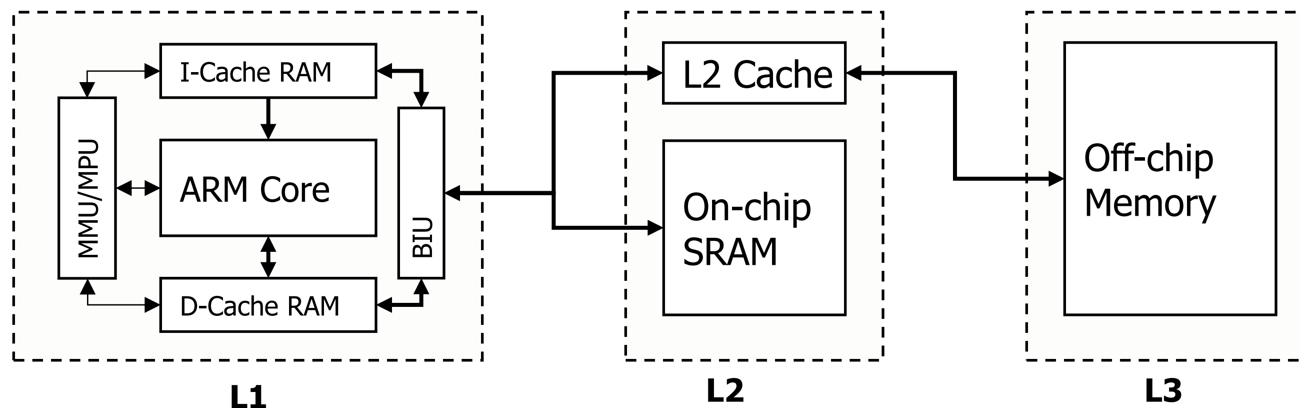


3. Map each unrolled operation onto a NEON vector lane, and generate corresponding NEON instructions

Processor Memory Map

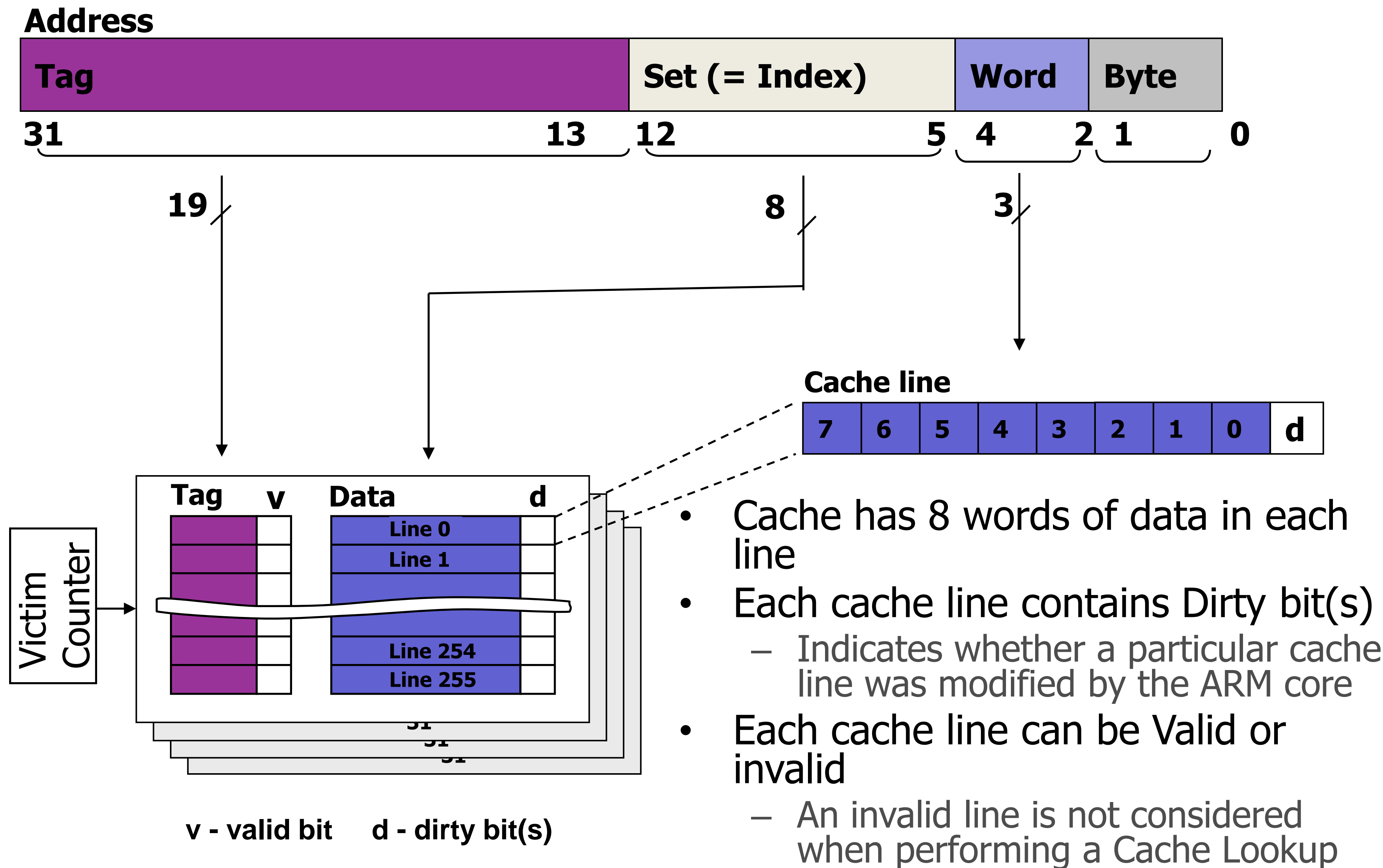


L1 and L2 Caches



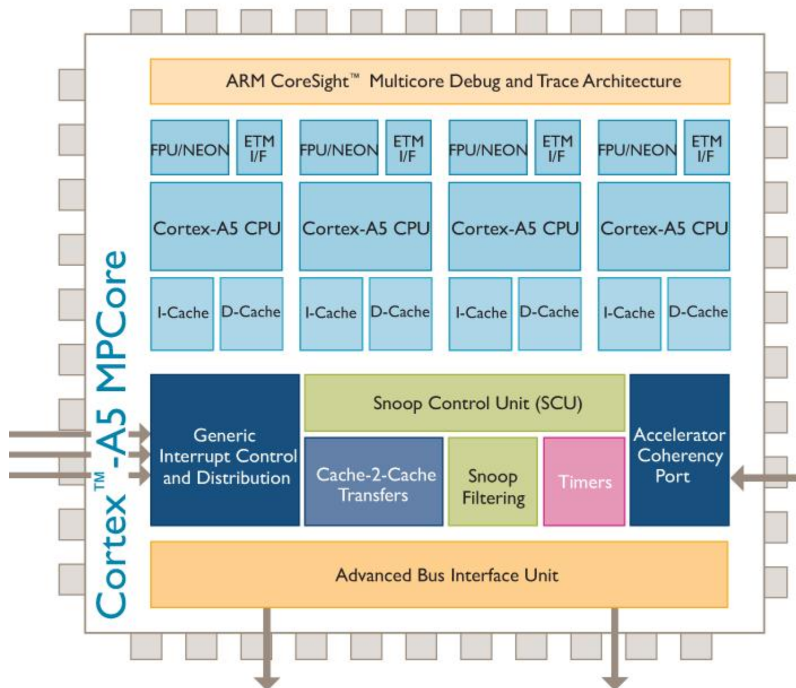
- Typical memory system can have multiple levels of cache
  - Level 1 memory system typically consists of L1-caches, MMU/MPU and TCMs
  - Level 2 memory system (and beyond) depends on the system design
- Memory attributes determine cache behavior at different levels
  - Controlled by the MMU/MPU (discussed later)
  - Inner Cacheable attributes define memory access behavior in the L1 memory system
  - Outer Cacheable attributes define memory access behavior in the L2 memory system (if external) and beyond (as signals on the bus)
- Before caches can be used, software setup must be performed

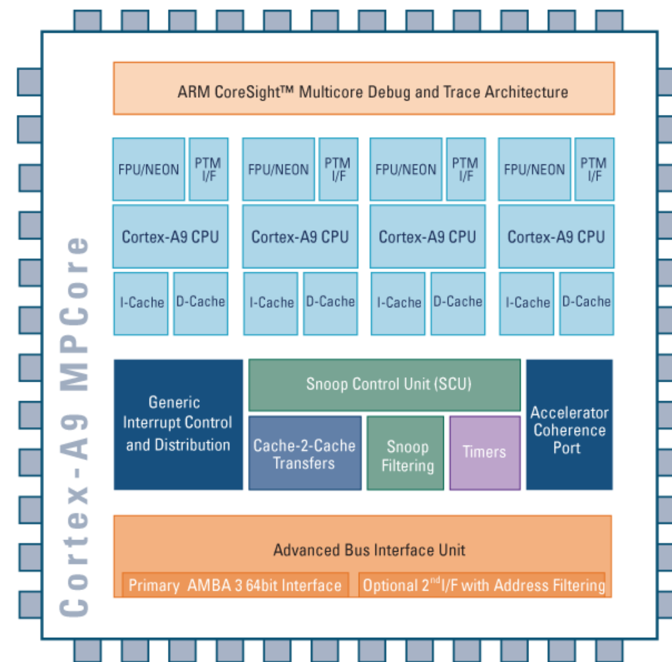
Example 32KB ARM cache



Cortex MPCore Processors

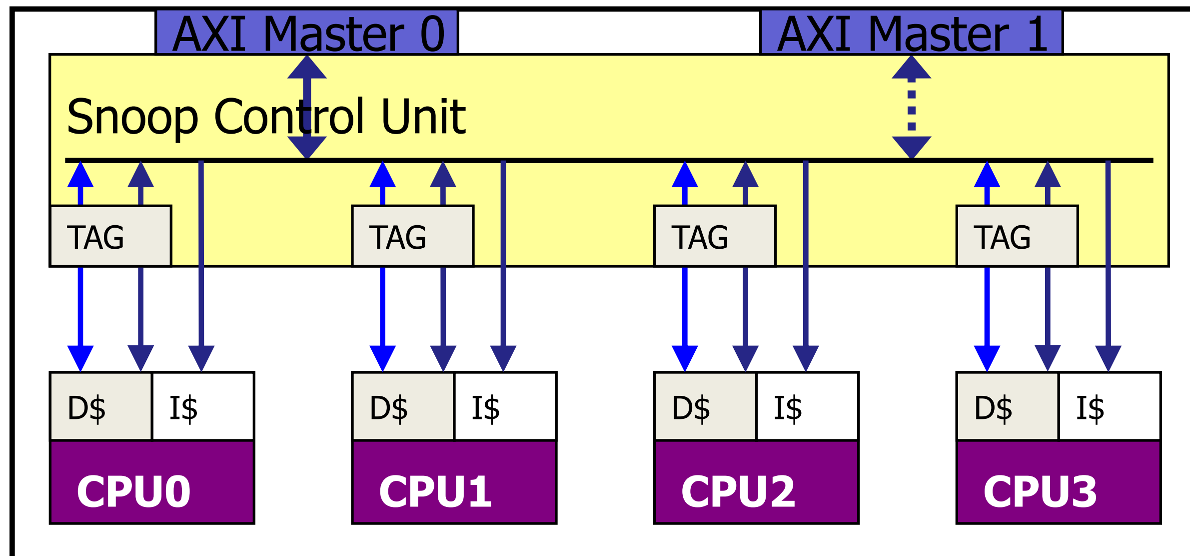
- Standard Cortex cores, with additional logic to support MPCore
  - Available as 1-4 CPU variants
- Include integrated
  - Interrupt controller
  - Snoop Control Unit (SCU)
  - Timers and Watchdogs





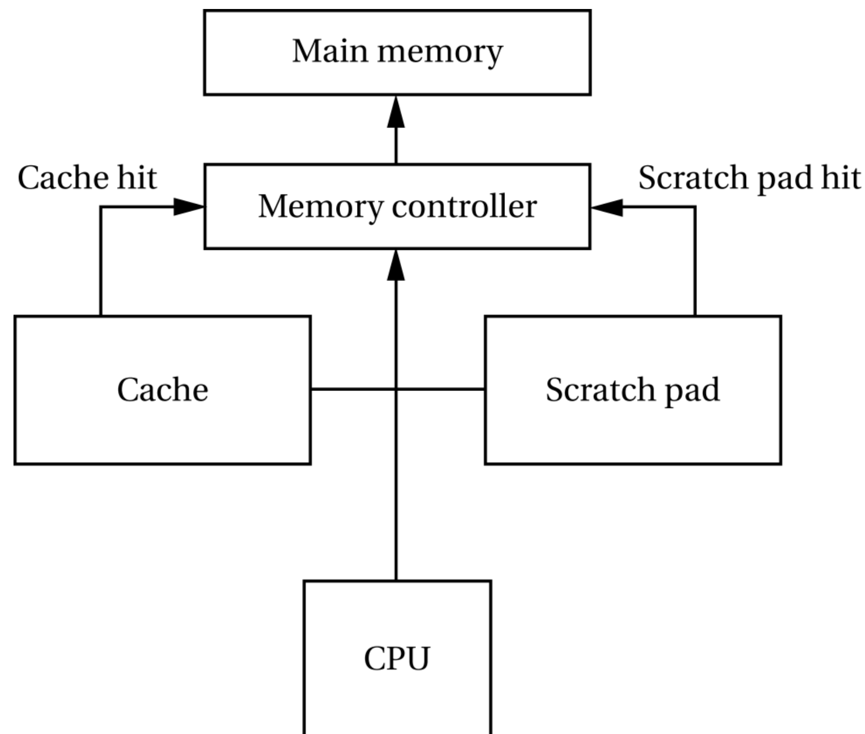
Snoop Control Unit

- The Snoop Control Unit (SCU) maintains coherency between L1 data caches
  - Duplicated Tag RAMs keep track of what data is allocated in each CPU's cache
    - Separate interfaces into L1 data caches for coherency maintenance
  - Arbitrates accesses to L2 AXI master interface(s), for both instructions and data
- Optionally, can use address filtering
  - Directing accesses to configured memory range to AXI Master port 1



Scratchpad Memory

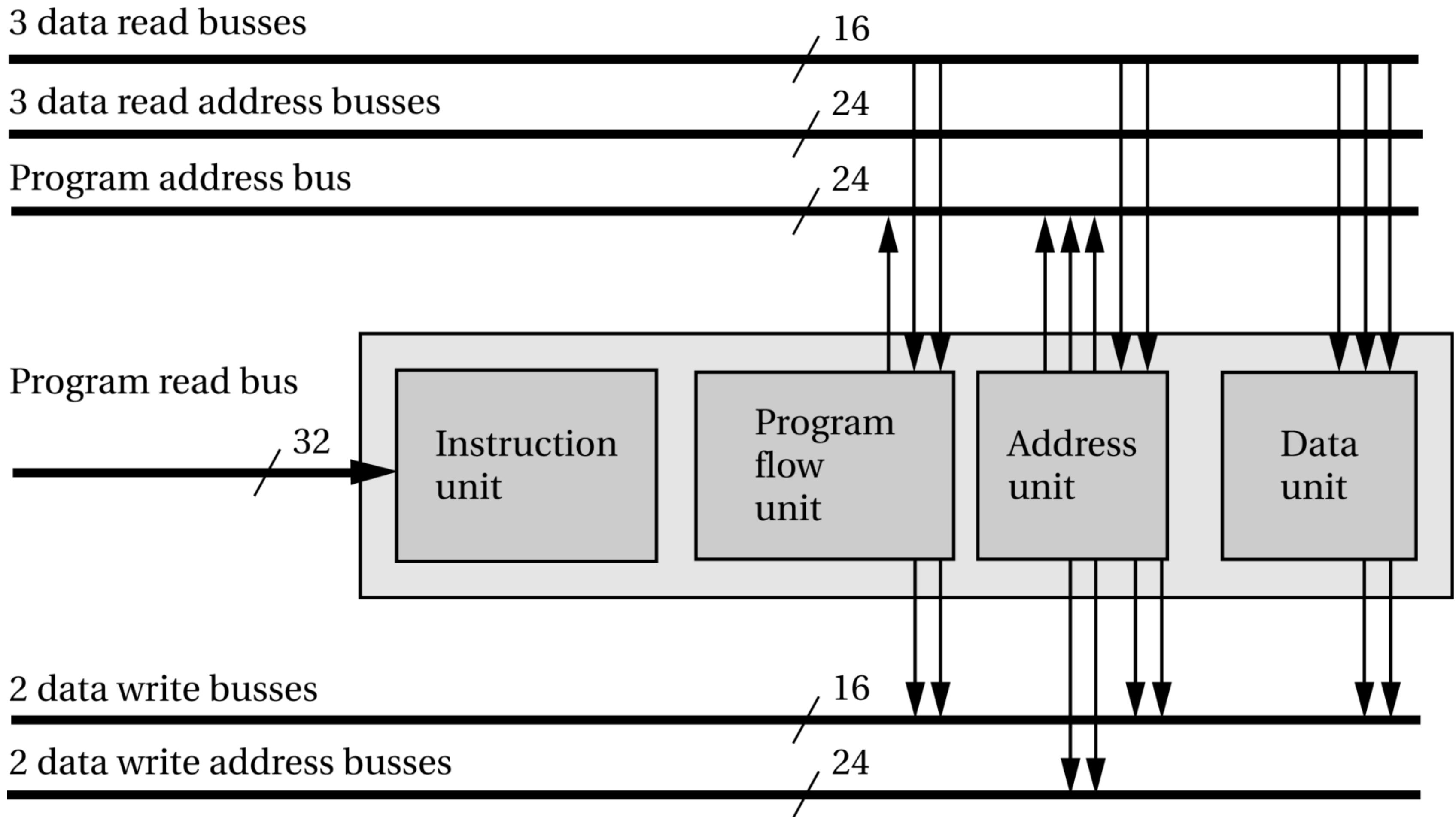
- Scratch pad is managed by software, not hardware
  - Provides predictable access time
  - Requires values to be allocated
- Use standard read/write instructions to access scratch pad



Digital Signal Processors

- First DSP was AT&T DSP16:
  - Hardware multiply-accumulate unit
  - Harvard architecture
- Today, DSP is often used as a marketing term
- Ex: TI C55x DSPs:
  - 40-bit arithmetic unit (32-bit values with 8 guard bits)
  - Barrel shifter
  - 17 x 17 multiplier
  - Comparison unit for Viterbi encoding/decoding
  - Single-cycle exponent encoder for wide-dynamic-range arithmetic
  - Two address generators

TI C55x Microarchitecture



TI C55x Overview

- Accumulator architecture:
  - acc = operand op acc.
  - Very useful in loops for DSP.
- C55x assembly language:

```
        MPY *AR0, *CDP+, AC0
Label:  MOV #1, T0
```

- C55x algebraic assembly language:

```
AC1 = AR0 * coef(*CDP)
```

Intrinsic Functions

- Compiler support for assembly language
- Intrinsic function maps directly onto an instruction
- Example:
  - int_sadd(arg1,arg2)
  - Performs saturation arithmetic addition

C55x Registers

- Terminology:
  - Register: any type of register
  - Accumulator: acc = operand op ac
- Most registers are memory-mapped
- Control-flow registers:
  - PC is program counter
  - XPC is program counter extension
  - RETA is subroutine return address

C55x Accumulators and Status Registers

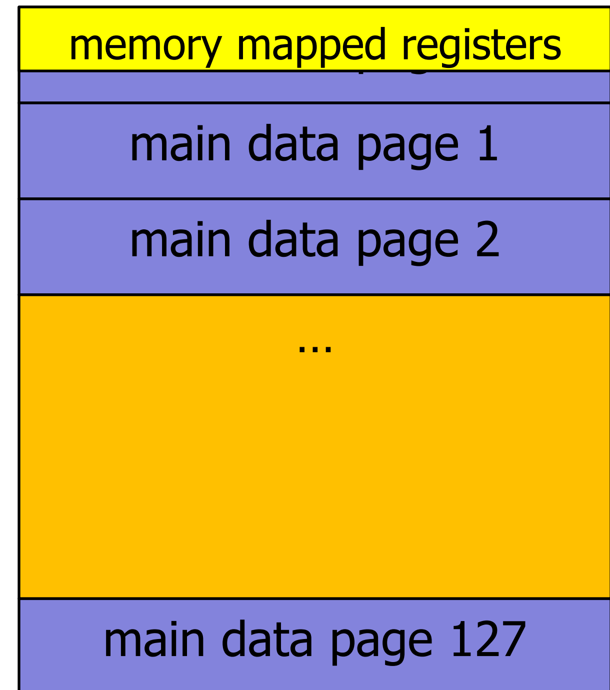
- Four 40-bit accumulators: AC0, AC1, AC2, and AC3
  - Low-order bits 0-15 are AC0L, etc
  - High-order bits 16-31 are AC0H, etc
  - Guard bits 32-39 are AC0G, etc
- ST0, ST1, PMST, ST0_55, ST1_55, ST2_55, ST3_55 provide arithmetic/bit manipulation flags, etc

C55x Auxiliary Registers

- AR0-AR7 are auxiliary registers
- CDP points to coefficients for polynomial evaluation instructions
  - CDPH is main data page pointer
- BK47 is used for circular buffer operations along with AR4-7
- BK03 addresses circular buffers
- BKC is size register for CDP

C55x Memory Map

- 24-bit address space, 16 MB of memory
- Data, program, I/O all mapped to same physical memory
- Addressability:
  - Program space address is 24 bits
  - Data space is 23 bits
  - I/O address is 16 bits



C55x Addressing Modes

- Three addressing modes:
  - Absolute addressing supplies an address in an instruction
  - Direct addressing supplies an offset
  - Indirect addressing uses a register as a pointer

C55x Data Operations

- MOV moves data between registers and memory:
  - MOV src, dst
- Varieties of ADDs:
  - ADD src,dst
  - ADD dual(LMEM),ACx,ACy
- Multiplication:
  - MPY src,dst
  - MAC AC,TX,ACy

C55x Control Flow

- Unconditional branch:
  - B ACx
  - B label
- Conditional branch:
  - BCC label, cond
- Loops:
  - Single-instruction repeat
  - Block repeat

Efficient Loops

- General rules:
  - Don't use function calls
  - Keep loop body small to enable local repeat (only forward branches)
  - Use unsigned integer for loop counter
  - Use <= to test loop counter
  - Make use of compiler---global optimization, software pipelining

Single-Instruction Loop Example

```
STM #4000h,AR2 ; load pointer to source
STM #100h,AR3 ; load pointer to destination
RPT #(1024-1)
MVDD *AR2+,*AR3+ ; move
```

C55x subroutines

- Unconditional subroutine call:
  - CALL target
- Conditional subroutine call:
  - CALLCC adrs,cond
- Two types of return:
  - Fast return gives return address and loop context in registers.
  - Slow return puts return address/loop on stack.

Acknowledgments

- These slides are inspired in part by material developed and copyright by:
  - Marilyn Wolf (Georgia Tech)
  - Steve Furber (University of Manchester)
  - William Stallings
  - ARM University Program