# CprE 488 – Embedded Systems Design

# Lecture 3 – Processors and Memory

Phillip Jones

Electrical and Computer Engineering

Iowa State University

[www.ece.iastate.edu/~zambreno](http://www.ece.iastate.edu/~zambreno)

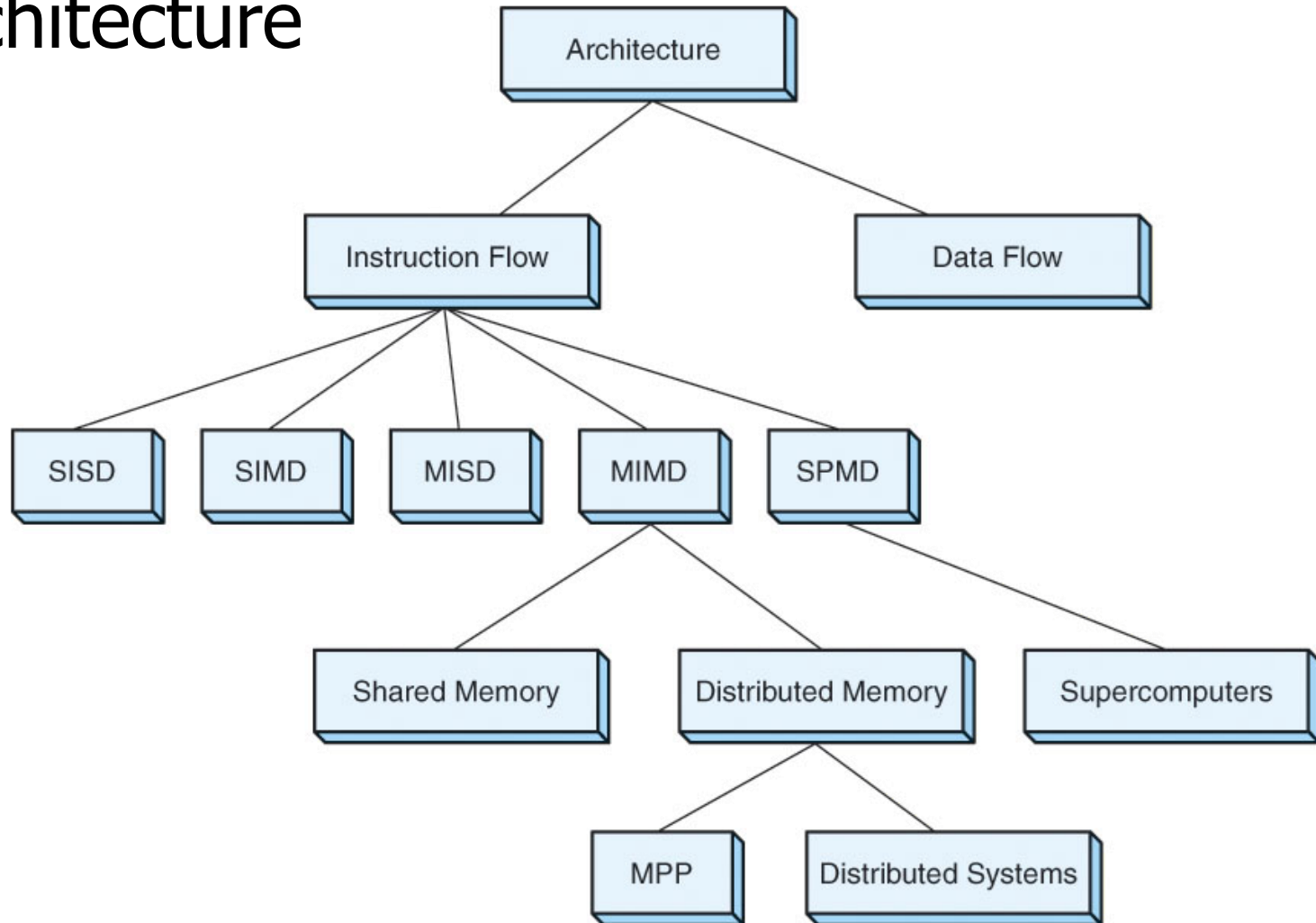[rcl.ece.iastate.edu](http://rcl.ece.iastate.edu)

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

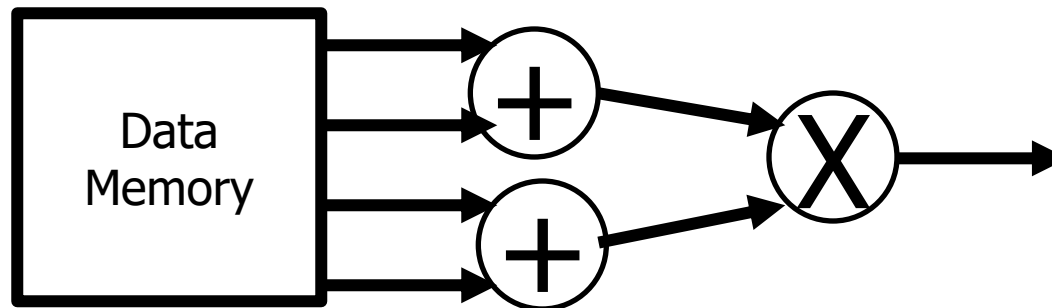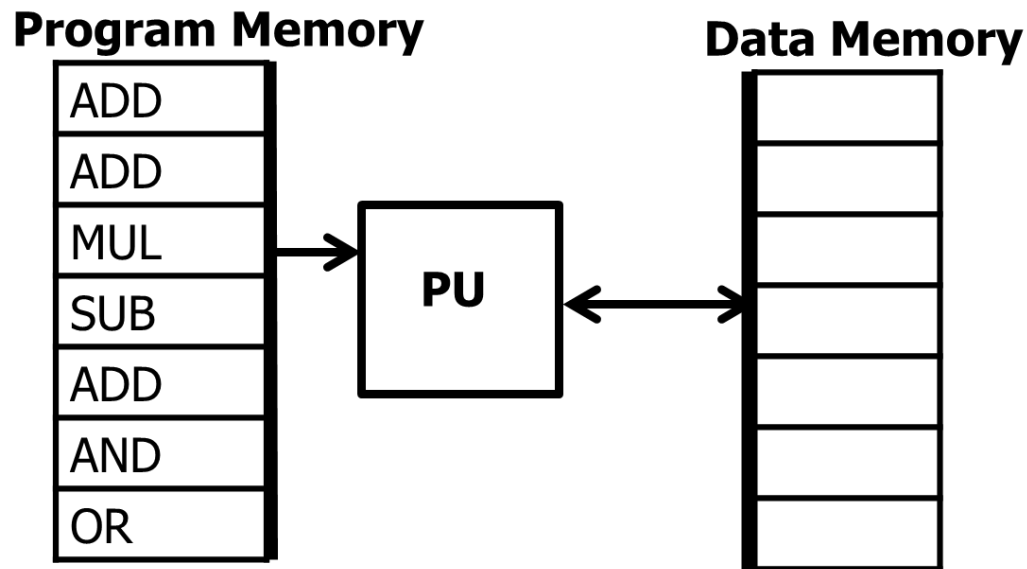# This Week's Topic

- Embedded processor design tradeoffs

- ISA and programming models

- Memory system mechanics

- Case studies:
  – ARM v7 CPUs  (RISC)
  – TI C55x DSPs  (CISC)
  – TI C64x  (VLIW)

- Reading: Wolf –
  – Chapter 2 (Instruction Sets),
  – Chapter 3.5 (Memory System Mechanisms)

# Flynn's (Updated) Taxonomy

- AKA the "alphabet soup" of computer architecture

# Flynn's Taxonomy: Instruction vs Data Flow

**Program Memory**

| ADD |
|-----|
| ADD |
| MUL |
| SUB |
| ADD |
| AND |
| OR  |

**PU**

**Data Memory**

Data Memory → (+) → (X) →

## Example: Custom Hardware for LQR Controller
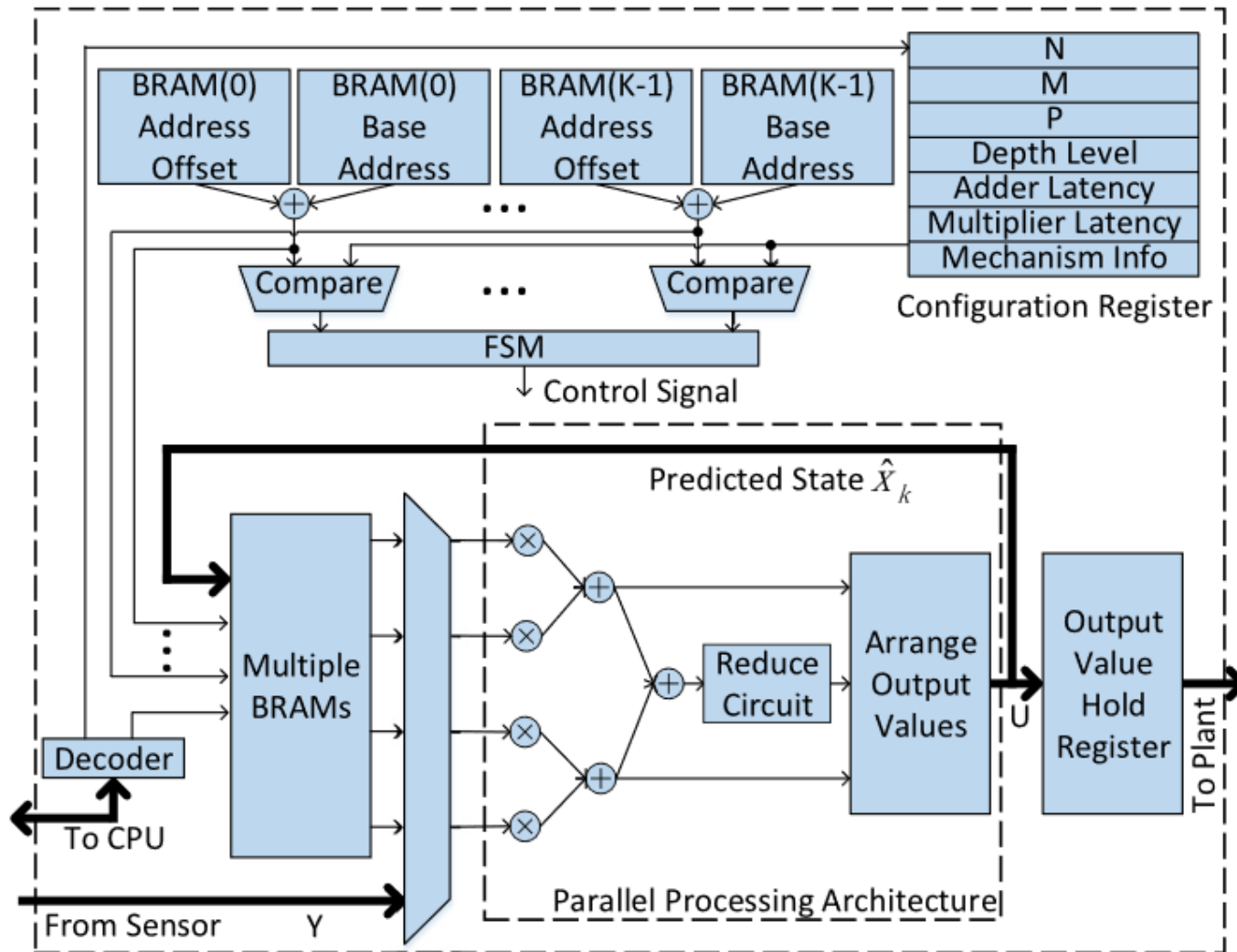
# Flynn's Taxonomy: Data Flow

Vector Matrix multiplication, primary computation of LQR controller

# Flynn's (Updated) Taxonomy

- AKA the "alphabet soup" of computer architecture

Link: https://en.wikipedia.org/wiki/Flynn%27s_taxonomy
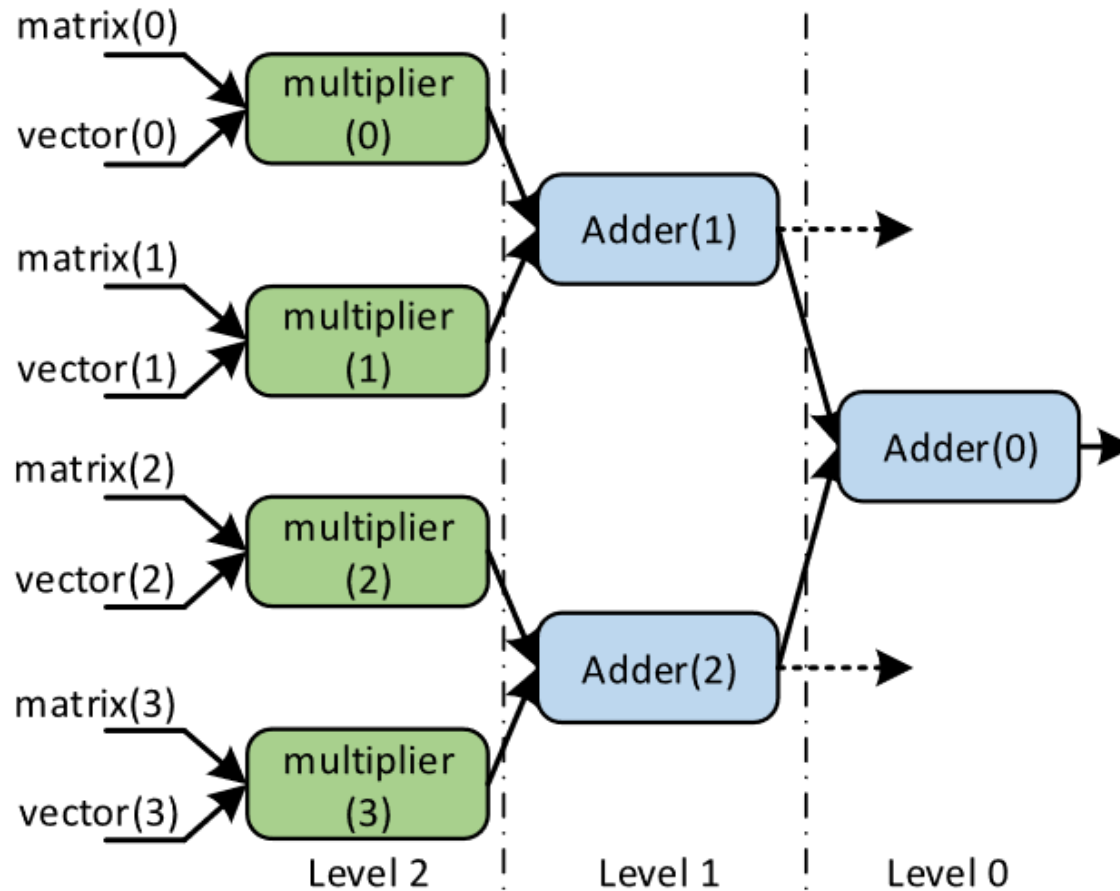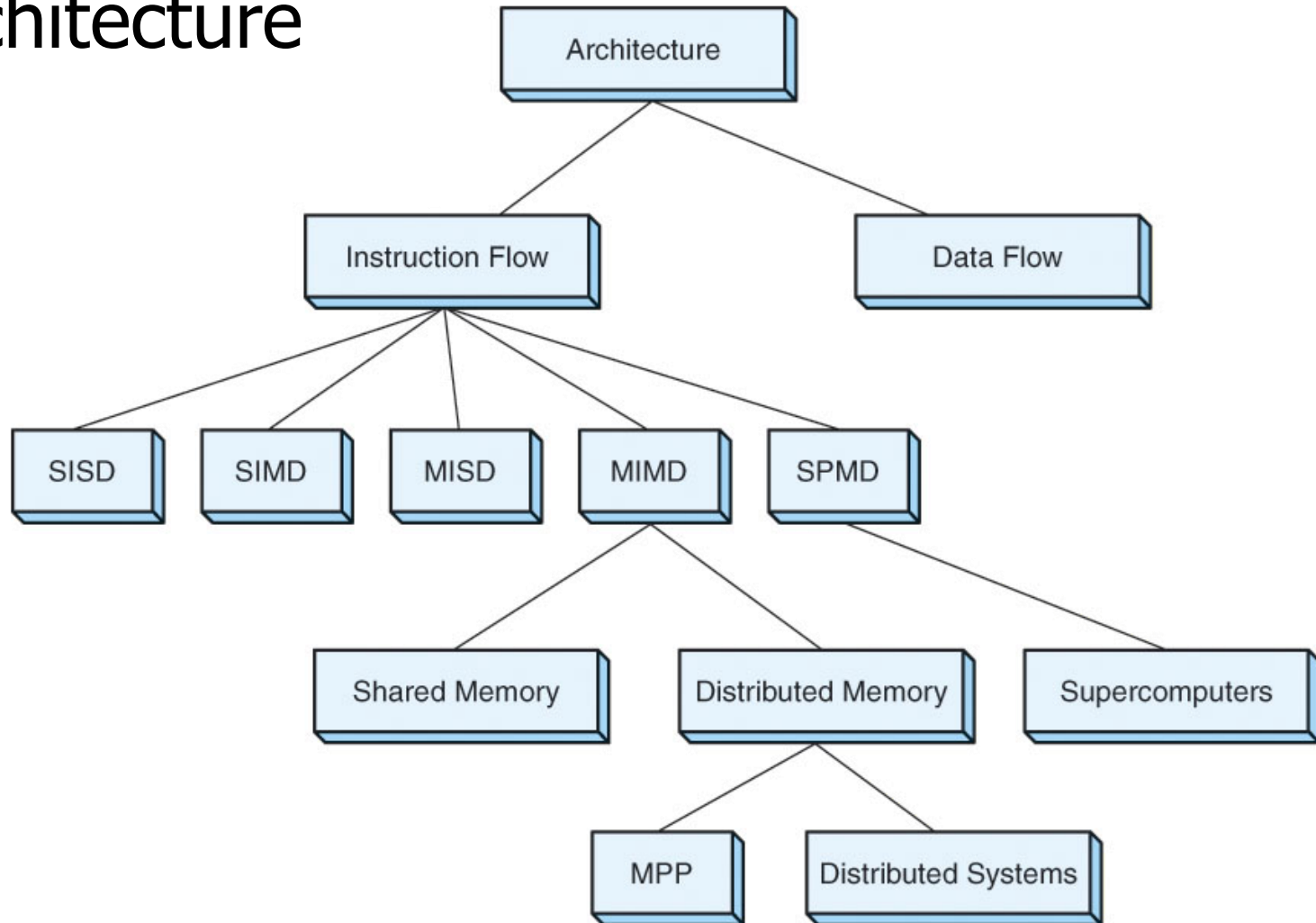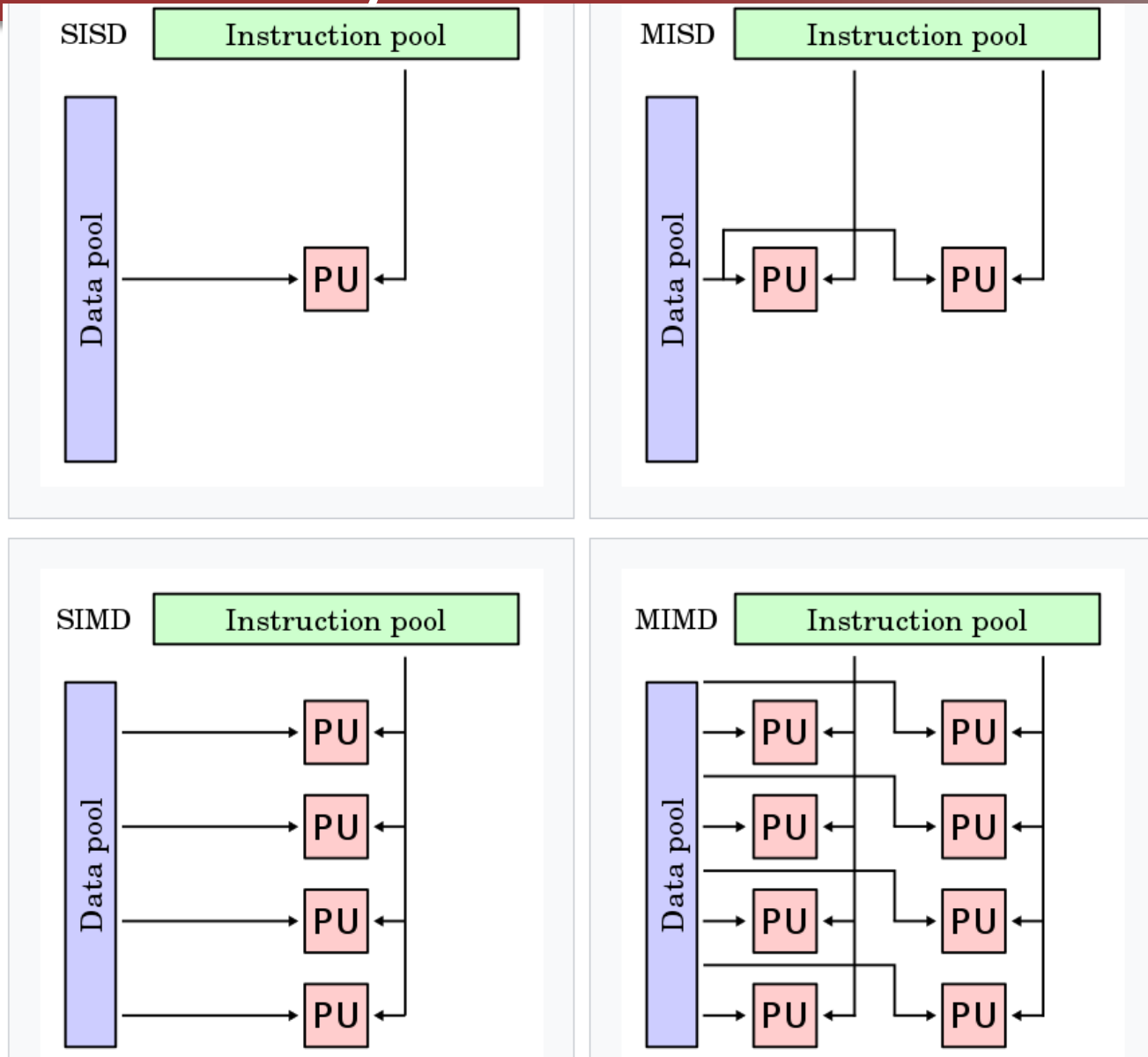
# ARM Architecture Revisions

| v4 | v5 | v6 | v7 |
|---|---|---|---|
| Halfword and signed halfword / byte support | Improved interworking | SIMD Instructions | Thumb-2 |
| | CLZ | Multi-processing | |
| System mode | Saturated arithmetic | v6 Memory architecture | Architecture Profiles |
| | DSP MAC instructions | Unaligned data support | 7-A - Applications |
| Thumb instruction set (v4T) | | Extensions: | 7-R - Real-time |
| | Extensions: | Thumb-2 (6T2) | 7-M - Microcontroller |
| | Jazelle (5TEJ) | TrustZone® (6Z) | |
| | | Multicore (6K) | |
| | | Thumb only (6-M) | |

- ## 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

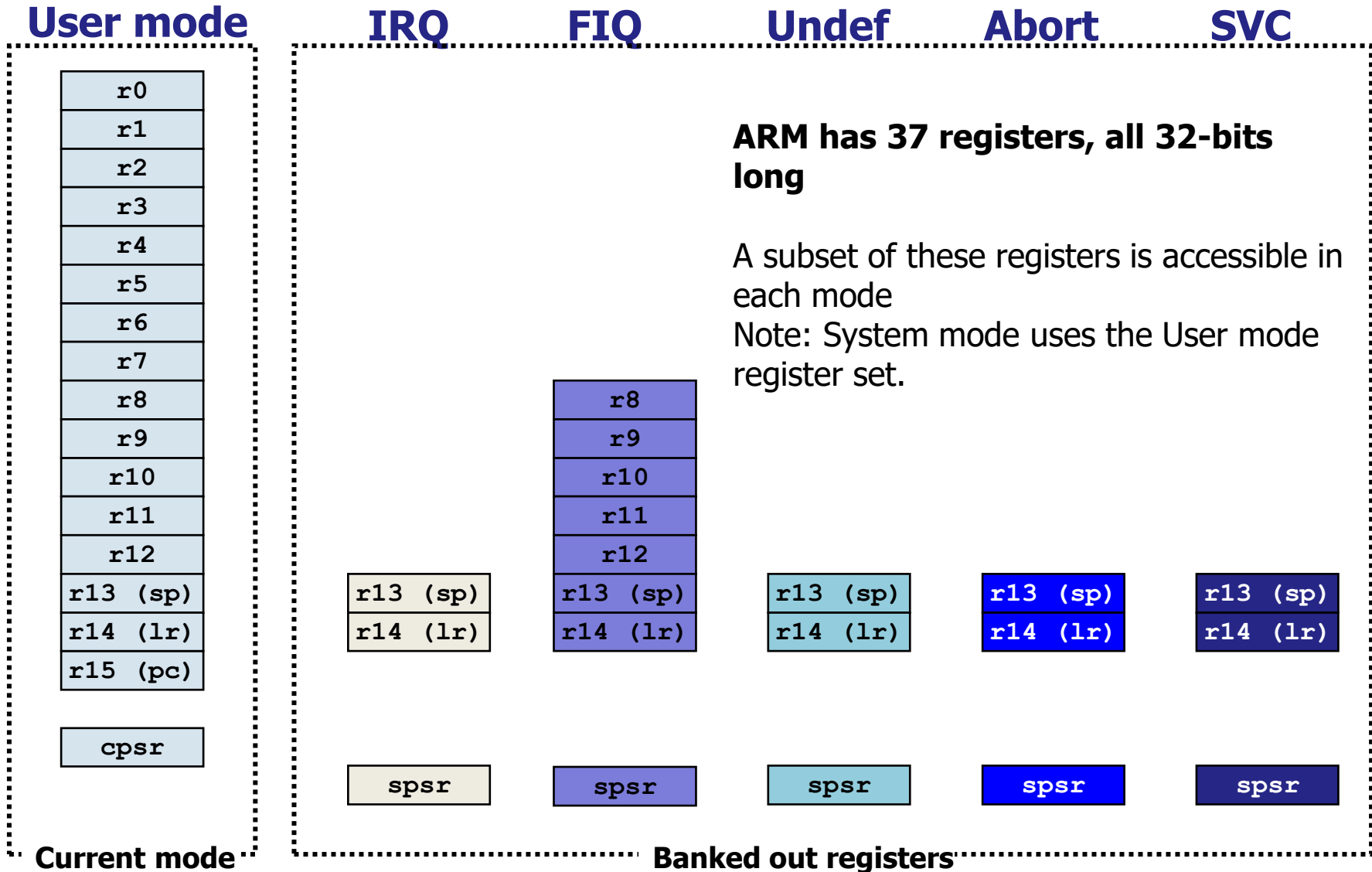- ARM cores implement two basic instruction sets
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set

- 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

| User mode | IRQ | FIQ | Undef | Abort | SVC |
|-----------|-----|-----|-------|-------|-----|

**ARM has 37 registers, all 32-bits long**

A subset of these registers is accessible in each mode
Note: System mode uses the User mode register set.

| User mode |
|-----------|
| r0 |
| r1 |
| r2 |
| r3 |
| r4 |
| r5 |
| r6 |
| r7 |
| r8 |
| r9 |
| r10 |
| r11 |
| r12 |
| r13 (sp) |
| r14 (lr) |
| r15 (pc) |

| cpsr |
|------|

**FIQ**

| r8 |
| r9 |
| r10 |
| r11 |
| r12 |

| IRQ | FIQ | Undef | Abort | SVC |
|-----|-----|-------|-------|-----|
| r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) | r13 (sp) |
| r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) | r14 (lr) |

| spsr | spsr | spsr | spsr | spsr |

**Current mode**          **Banked out registers**

# Program Status Registers

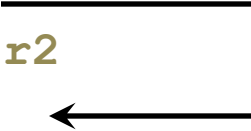| 31 | | 28 | 27 | | 24 | 23 | | 16 | 15 | | 8 | 7 | 6 | 5 | 4 | | 0 |
|----|---|----|----|---|----|----|---|----|----|---|---|---|---|---|---|---|---|
| N | Z | C | V | Q | | J | U n d e f | | | i n e d | | | I | F | T | mode | |
| **f** | | | | | | **s** | | | | **x** | | | **c** | | | | |
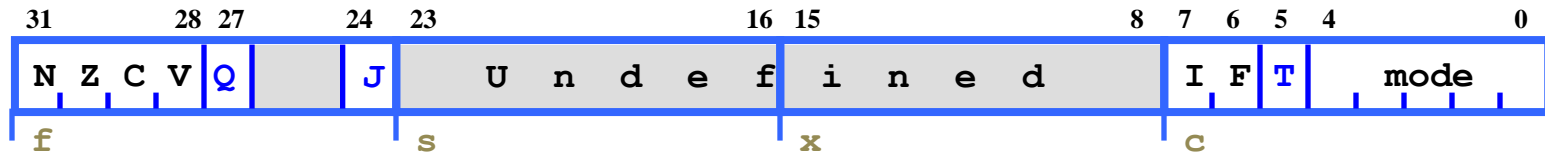
- **Condition code flags**
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed

- **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                              CMP    r3,#0
BEQ    skip                               ADDNE  r0,r1,r2
ADD    r0,r1,r2
skip
```
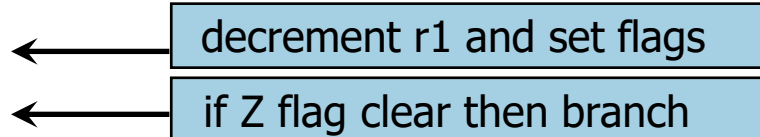
- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S".  CMP does not need "S".

```
loop

   …
SUBS r1,r1,#1          decrement r1 and set flags
BNE loop              if Z flag clear then branch
```

# 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
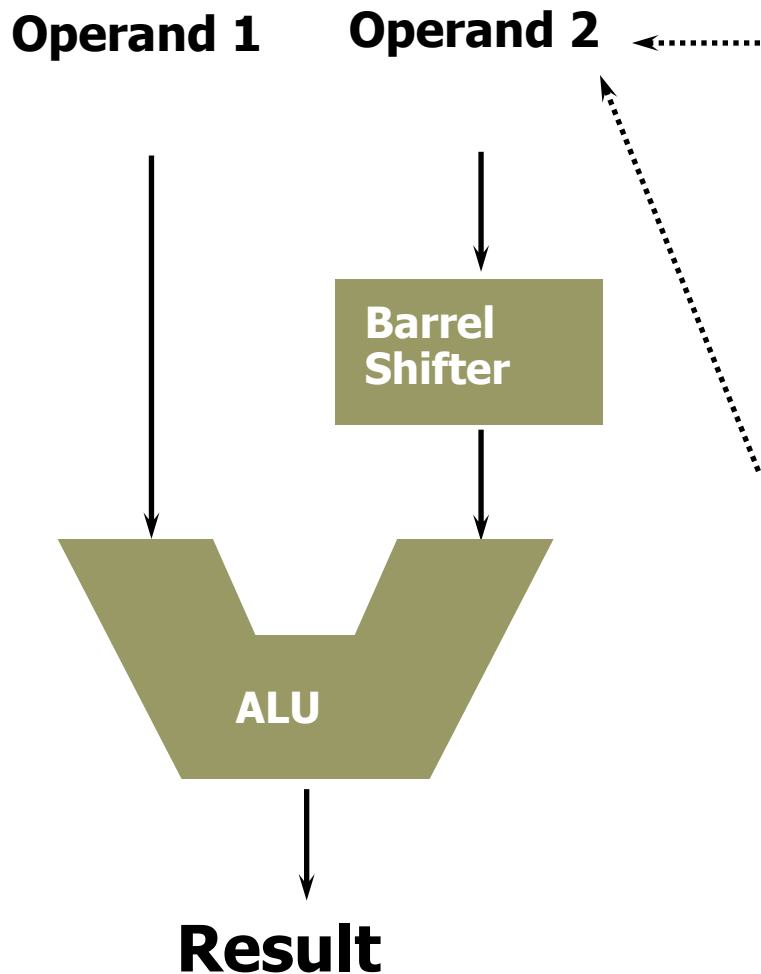
- Syntax:

    **<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

**Operand 1**      **Operand 2** ◄········

**Barrel Shifter**

**ALU**

**Result**

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 (i.e. in two instructions)

# 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

```
      11    8 7        0
_____
     |  rot  | immed_8 |
_____
    x2          |
     \___       ↓
         →  ┌─────────┐
            │ Shifter │
            │   ROR   │
            └─────────┘
                 ↓
```
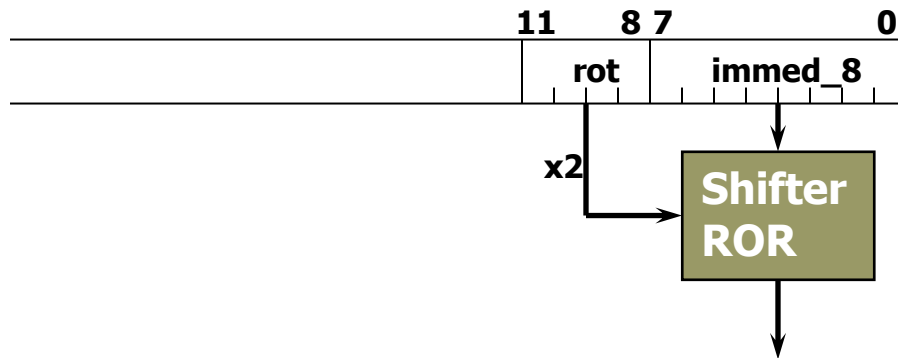
**Quick Quiz:**
`0xe3a004ff`
`MOV r0, #???`

- 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`
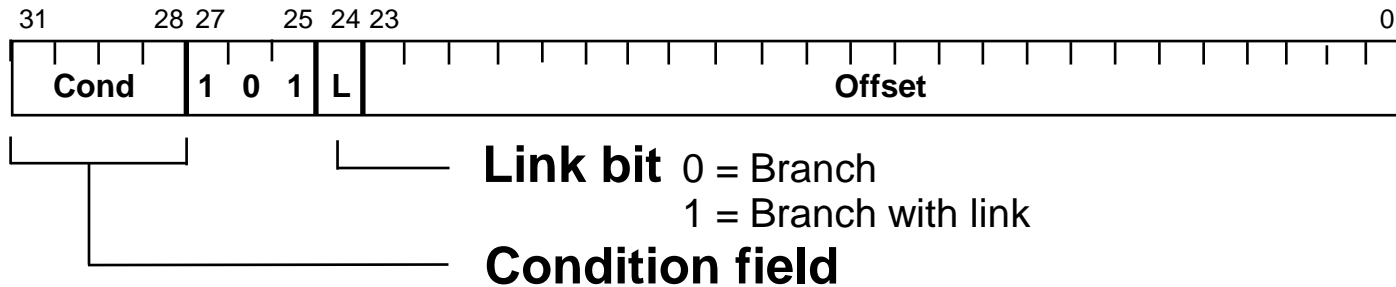
- 64-bit multiply instructions offer both signed and unsigned versions
  - For these instruction there are 2 destination registers

  - `[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`

```
31        28 27    25 24 23                                         0
┌──────────┬───────┬──┬────────────────────────────────────────────┐
│   Cond   │ 1  0  1│ L│                  Offset                     │
└──────────┴───────┴──┴────────────────────────────────────────────┘
```

**Link bit**  0 = Branch
              1 = Branch with link
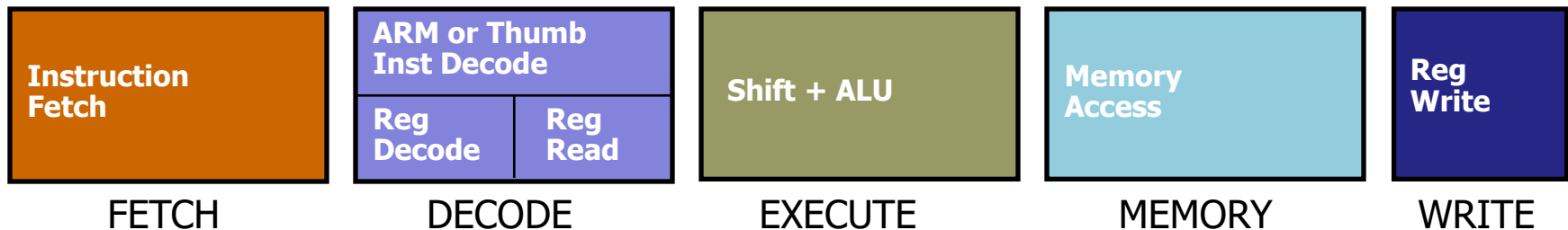**Condition field**

- 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

**ARM7TDMI**

| | | |
|---|---|---|
| Instruction Fetch | Thumb→ARM decompress / ARM decode, Reg Select | Reg Read, Shift, ALU, Reg Write |
| FETCH | DECODE | EXECUTE |

**ARM9TDMI**

| | | | | |
|---|---|---|---|---|
| Instruction Fetch | ARM or Thumb Inst Decode / Reg Decode, Reg Read | Shift + ALU | Memory Access | Reg Write |
| FETCH | DECODE | EXECUTE | MEMORY | WRITE |

# ARM Pipeline Evolution (cont.)

## ARM10

| FETCH | ISSUE | DECODE | EXECUTE | MEMORY | WRITE |
|---|---|---|---|---|---|
| Branch Prediction / Instruction Fetch | ARM or Thumb Instruction Decode | Reg Read | Shift + ALU / Multiply | Memory Access / Multiply Add | Reg Write |

## ARM11

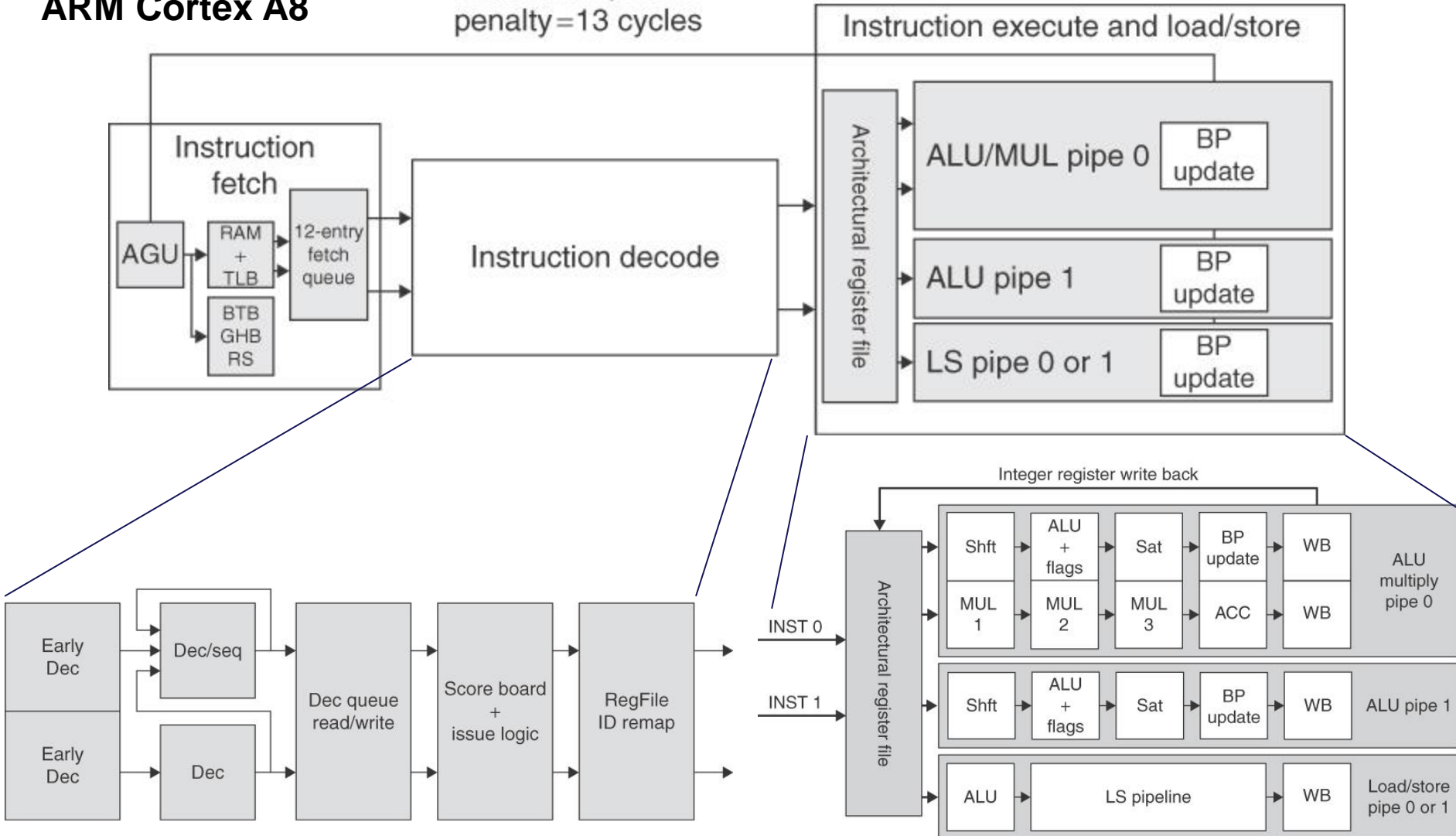| Fetch 1 | Fetch 2 | Decode | Issue | Shift | ALU | Saturate | Write back |
|---|---|---|---|---|---|---|---|
| | | | | MAC 1 | MAC 2 | MAC 3 | |
| | | | | Address | Data Cache 1 | Data Cache 2 | |

# ARM Pipeline Evolution (cont.)



F0  F1  F2    D0  D1  D2  D3  D4    E0  E1  E2  E3  E4  E5

**ARM Cortex A8**

Branch mispredict penalty=13 cycles

Instruction execute and load/store

Instruction fetch

AGU → RAM + TLB → 12-entry fetch queue

BTB GHB RS

Instruction decode

Architectural register file

ALU/MUL pipe 0 — BP update

ALU pipe 1 — BP update

LS pipe 0 or 1 — BP update

Early Dec → Dec/seq → Dec queue read/write → Score board + issue logic → RegFile ID remap

Early Dec → Dec

INST 0

INST 1

Architectural register file

Integer register write back

Shft → ALU + flags → Sat → BP update → WB   |   ALU multiply pipe 0
MUL 1 → MUL 2 → MUL 3 → ACC → WB

Shft → ALU + flags → Sat → BP update → WB   |   ALU pipe 1

ALU → LS pipeline → WB   |   Load/store pipe 0 or 1

# 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)
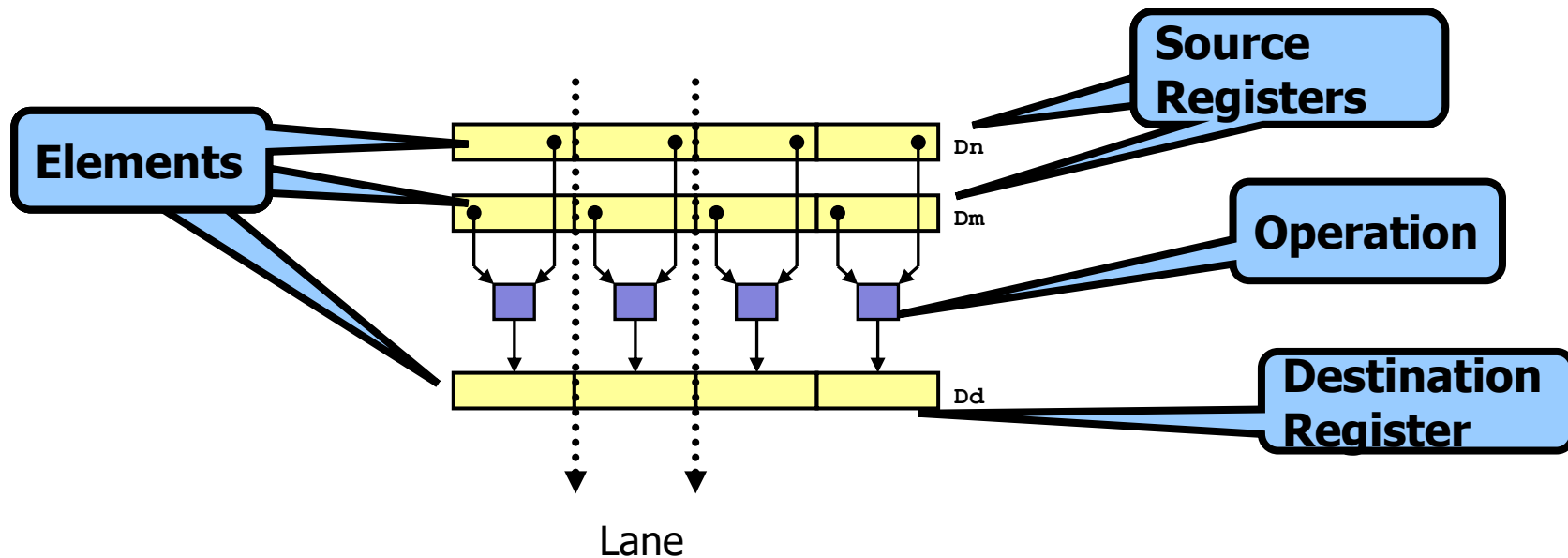


- NEON instructions perform "Packed SIMD" processing
  - Registers are considered as **vectors** of **elements** of the same data type
  - Data types available: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions usually perform the same operation in all **lanes**

# 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)

| D0 |
| --- |
| D1 |
| D2 |
| D3 |
| : |
| D30 |
| D31 |

........

| .... Q0 ..... |
| --- |
| .... Q1 ..... |
| : |
| .... Q15 ..... |
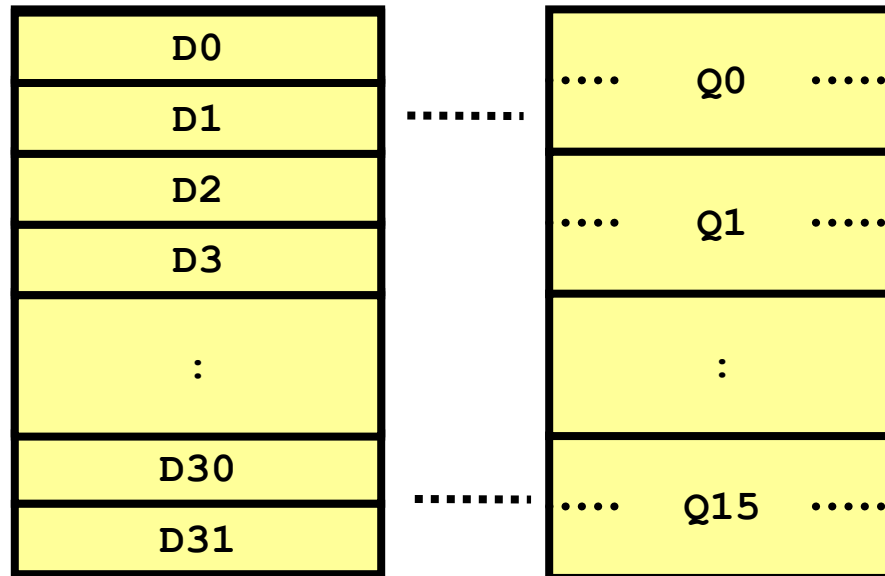
- Enables register trade-offs
  - Vector length can be variable
  - Different registers available

# 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;
}
```
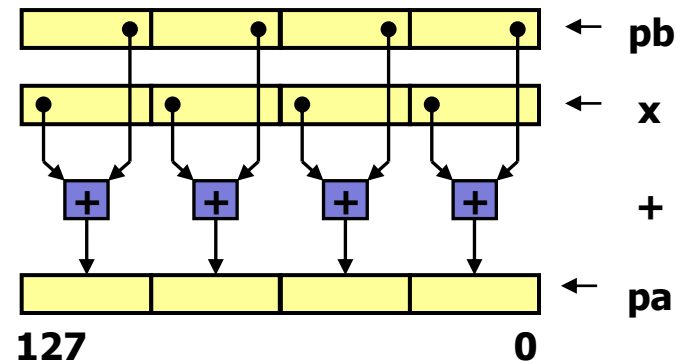
1. Analyze each loop:

   - Are pointer accesses safe for vectorization?

   - What data types are being used? How do they map onto NEON vector registers?
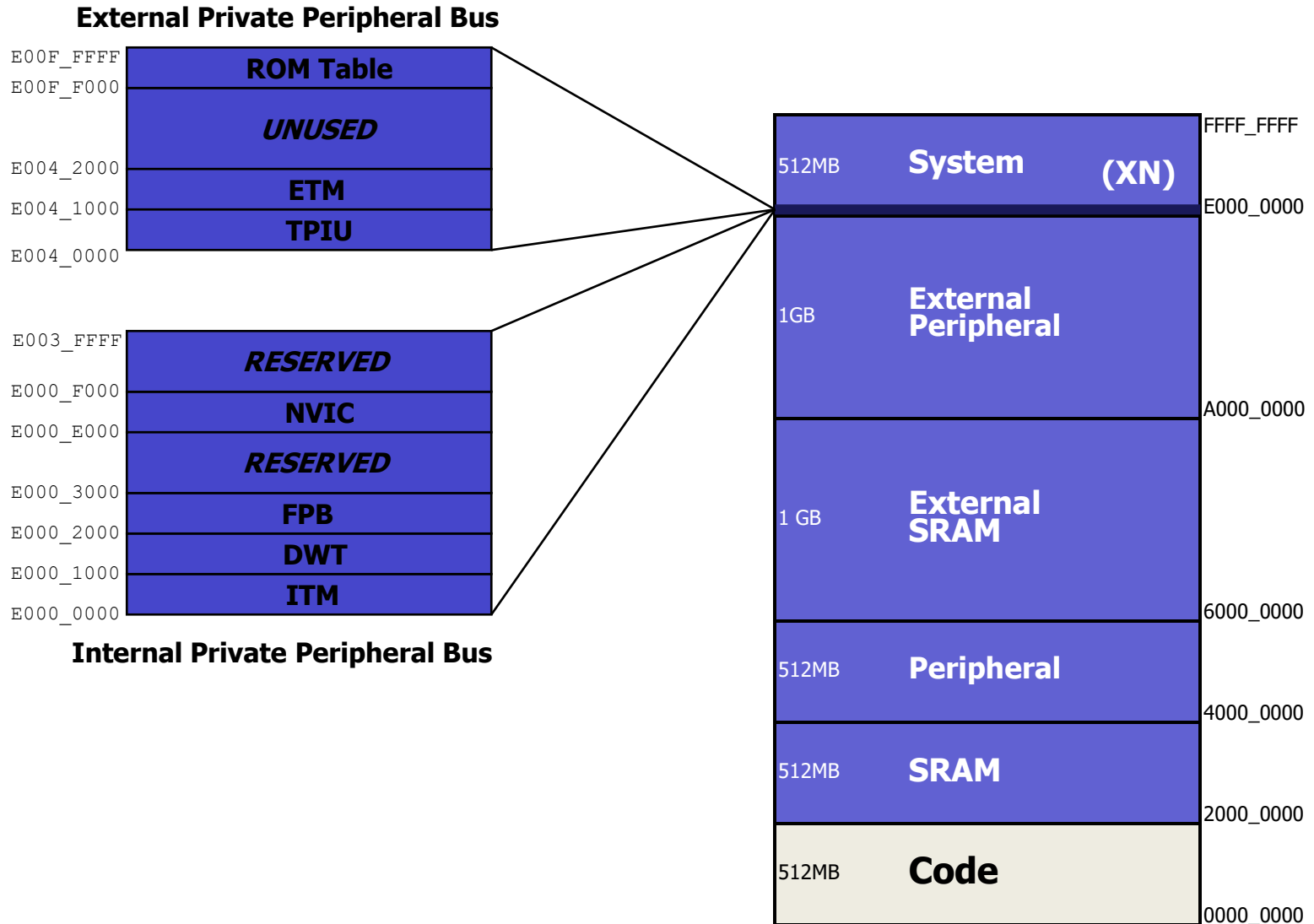
   - Number of loop iterations

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

2. Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization
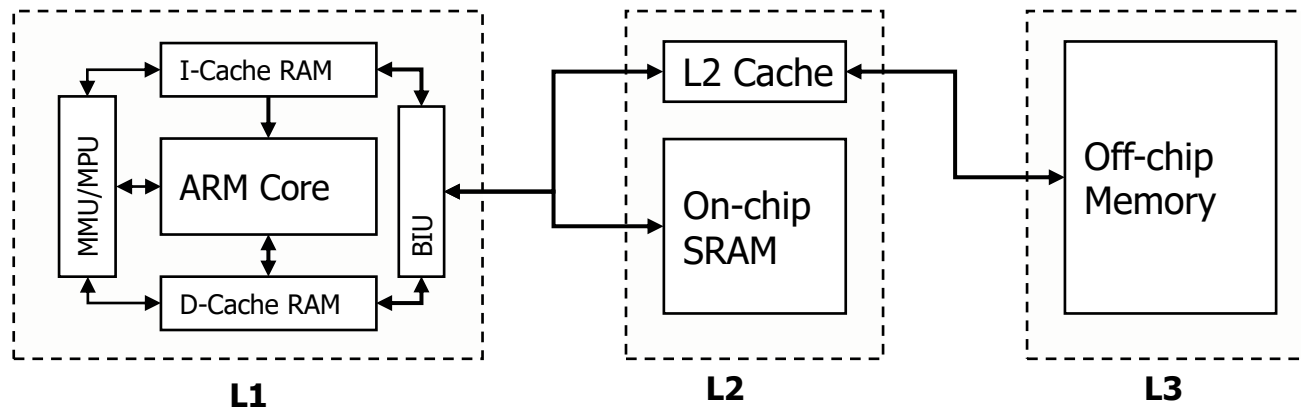
```
void add_int(int *pa, int *pb,
             unsigned n, int x)
{
  unsigned int i;
  for (i = ((n & ~3) >> 2); i; i--)
  {
    *(pa + 0) = *(pb + 0) + x;
    *(pa + 1) = *(pb + 1) + x;
    *(pa + 2) = *(pb + 2) + x;
    *(pa + 3) = *(pb + 3) + x;
    pa += 4; pb += 4;
  }
}
```

# Processor Memory Map

**External Private Peripheral Bus**

| Address | Region |
|---|---|
| E00F_FFFF | ROM Table |
| E00F_F000 | |
| | *UNUSED* |
| E004_2000 | |
| E004_1000 | ETM |
| E004_0000 | TPIU |

**Internal Private Peripheral Bus**

| Address | Region |
|---|---|
| E003_FFFF | *RESERVED* |
| E000_F000 | NVIC |
| E000_E000 | |
| | *RESERVED* |
| E000_3000 | FPB |
| E000_2000 | DWT |
| E000_1000 | ITM |
| E000_0000 | |

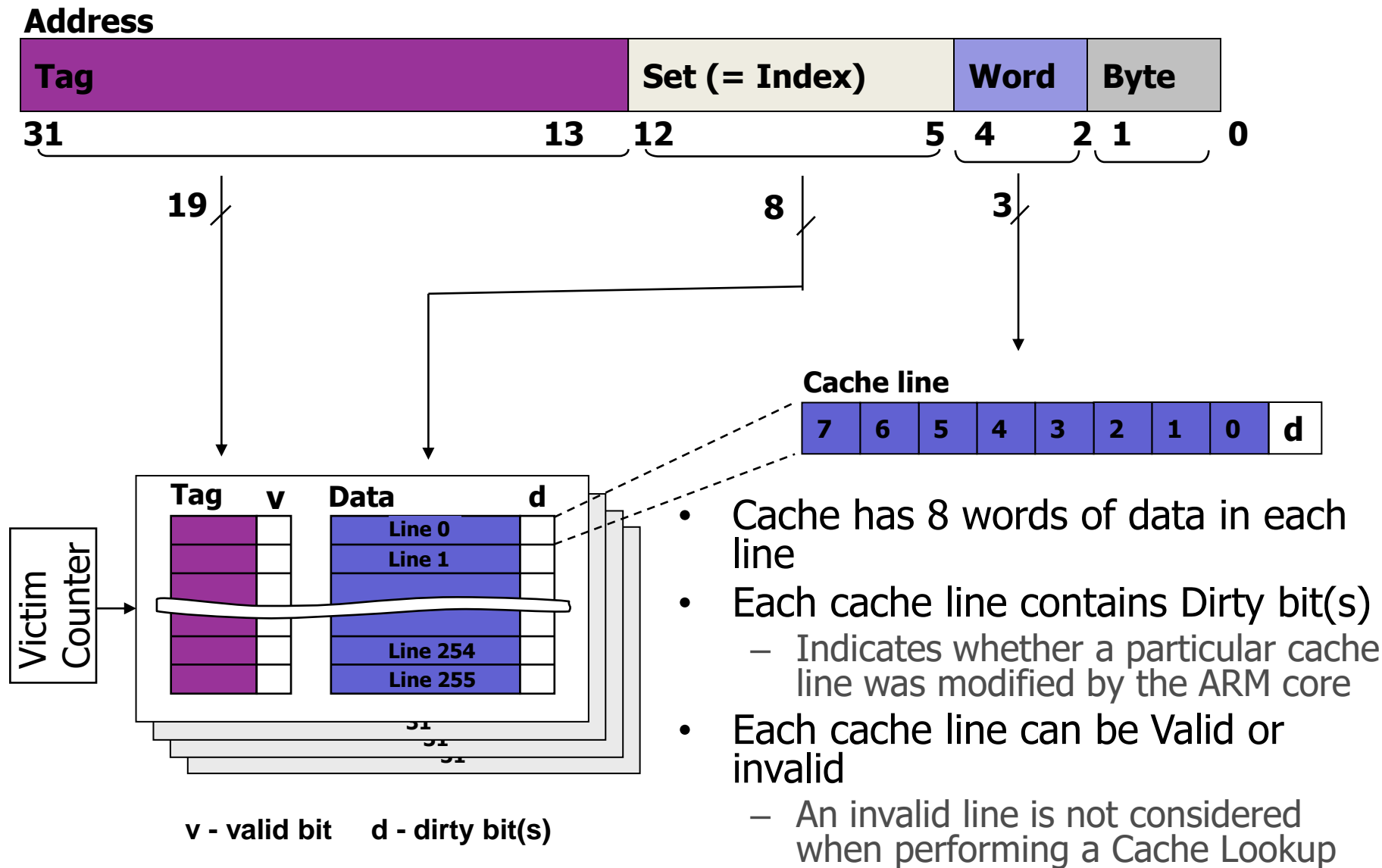| Size | Region | Address |
|---|---|---|
| 512MB | System **(XN)** | FFFF_FFFF / E000_0000 |
| 1GB | External Peripheral | A000_0000 |
| 1 GB | External SRAM | 6000_0000 |
| 512MB | Peripheral | 4000_0000 |
| 512MB | SRAM | 2000_0000 |
| 512MB | **Code** | 0000_0000 |

# 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

**Address**

| Tag | Set (= Index) | Word | Byte |
|---|---|---|---|

31                      13   12             5   4   2  1   0

19          8          3

**Cache line**

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | d |
|---|---|---|---|---|---|---|---|---|

Victim Counter

| Tag | v | Data | d |
|---|---|---|---|
| | | Line 0 | |
| | | Line 1 | |
| | | Line 254 | |
| | | Line 255 | |

**v - valid bit**  **d - dirty bit(s)**

- Cache has 8 words of data in each line
- Each cache line contains Dirty bit(s)
  - Indicates whether a particular cache line was modified by the ARM core
- Each cache line can be Valid or invalid
  - An invalid line is not considered when performing a Cache Lookup

# 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
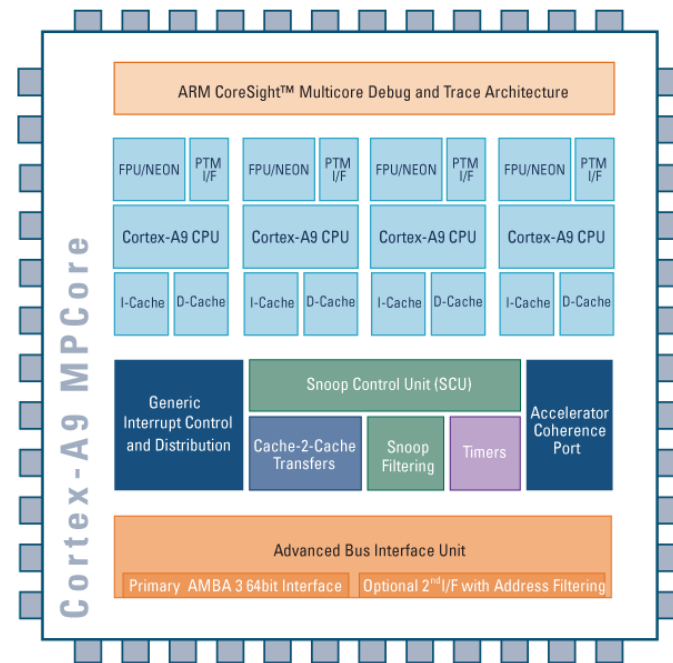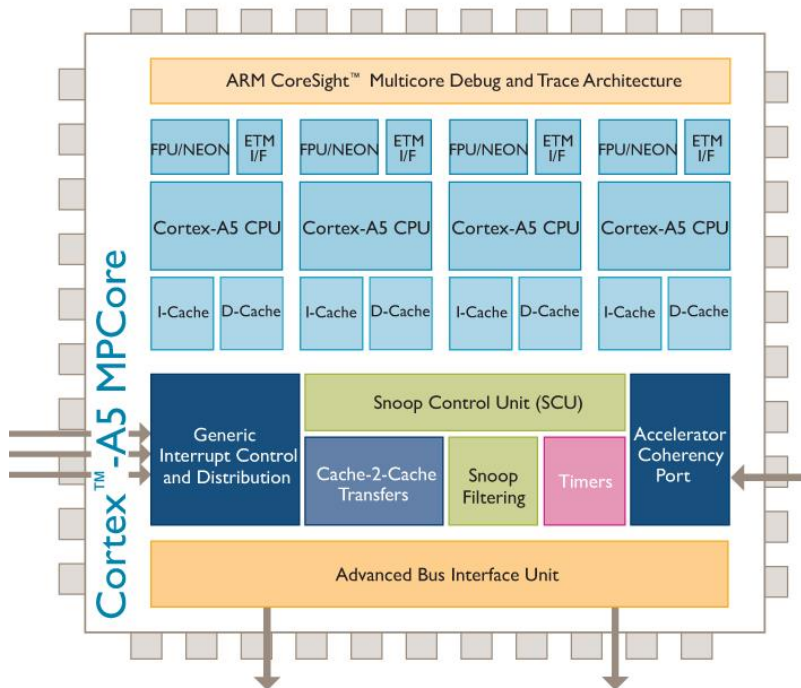
# Snoop Control Unit

- 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
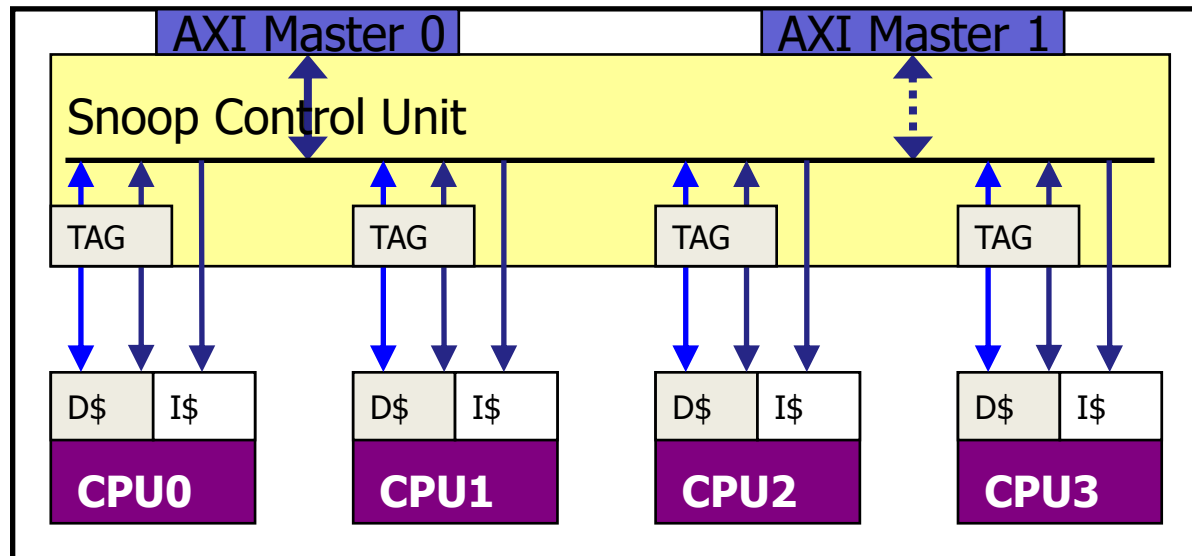
# Scratchpad Memory

- 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
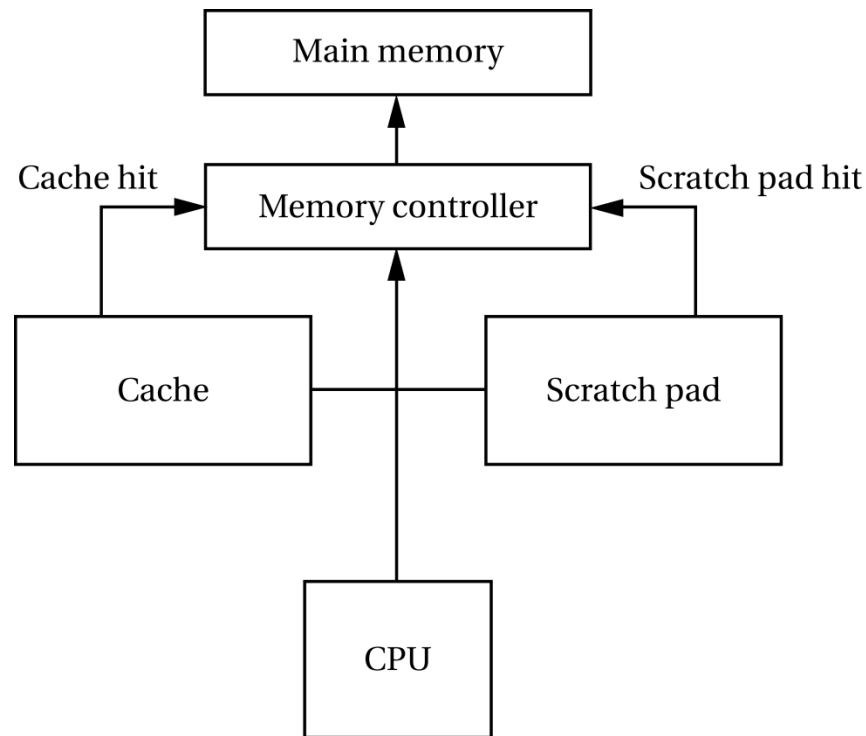
# Digital Signal Processors

- 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

# RISC vs CISC: Register Structure

**ARM**

| r0 |
| --- |
| r1 |
| r2 |
| r3 |
| r4 |
| r5 |
| r6 |
| r7 |
| r8 |
| r9 |
| r10 |
| r11 |
| r12 |
| r13 (sp) |
| r14 (lr) |
| r15 (pc) |

| cpsr |
| --- |

# RISC vs CISC: Register Structure

## **ARM**

General Purpose:
R0 – R15 (32-bit)

SP: R13    (32-bit)
LR: R14    (32-bit)
PC: R15    (32-bit)

Status Register:
CPSR       (32-bit)

# RISC vs CISC: Register Structure

## C55x (DSP)

## ARM

General Purpose:
R0 – R15 (32-bit)

SP: R13   (32-bit)
LR: R14   (32-bit)
PC: R15   (32-bit)

Status Register:
CPSR       (32-bit)

# RISC vs CISC: Register Structure

## C55x (DSP)

Table 2–1. Alphabetical Summary of Registers

| Register Name | Description | Size |
|---|---|---|
| AC0–AC3 | Accumulators 0 through 3 | 40 bits each |
| AR0–AR7 | Auxiliary registers 0 through 7 | 16 bits each |
| BK03, BK47, BKC | Circular buffer size registers | 16 bits each |
| BRC0, BRC1 | Block-repeat counters 0 and 1 | 16 bits each |
| BRS1 | BRC1 save register | 16 bits |
| BSA01, BSA23, BSA45, BSA67, BSAC | Circular buffer start address registers | 16 bits each |
| CDP | Coefficient data pointer (low part of XCDP) | 16 bits |
| CDPH | High part of XCDP | 7 bits |
| CFCT | Control-flow context register | 8 bits |
| CSR | Computed single-repeat register | 16 bits |
| DBIER0, DBIER1 | Debug interrupt enable registers 0 and 1 | 16 bits each |
| DP | Data page register (low part of XDP) | 16 bits |
| DPH | High part of XDP | 7 bits |

## ARM

General Purpose:
R0 – R15 (32-bit)

SP: R13   (32-bit)
LR: R14   (32-bit)
PC: R15   (32-bit)

Status Register:
CPSR      (32-bit)

# RISC vs CISC: Register Structure

## C55x (DSP)

| | | |
|---|---|---|
| IER0, IER1 | Interrupt enable registers 0 and 1 | 16 bits each |
| IFR0, IFR1 | Interrupt flag registers 0 and 1 | 16 bits each |
| IVPD, IVPH | Interrupt vector pointers | 16 bits each |
| PC | Program counter | 24 bits |
| PDP | Peripheral data page register | 9 bits |
| REA0, REA1 | Block-repeat end address registers 0 and 1 | 24 bits each |
| RETA | Return address register | 24 bits |
| RPTC | Single-repeat counter | 16 bits |
| RSA0, RSA1 | Block-repeat start address registers 0 and 1 | 24 bits each |
| SP | Data stack pointer (low part of XSP) | 16 bits |
| SPH | High part of XSP and XSSP | 7 bits |
| SSP | System stack pointer (low part of XSSP) | 16 bits |
| ST0_55–ST3_55 | Status registers 0 through 3 | 16 bits each |
| T0–T3 | Temporary registers | 16 bits each |
| TRN0, TRN1 | Transition registers 0 and 1 | 16 bits each |

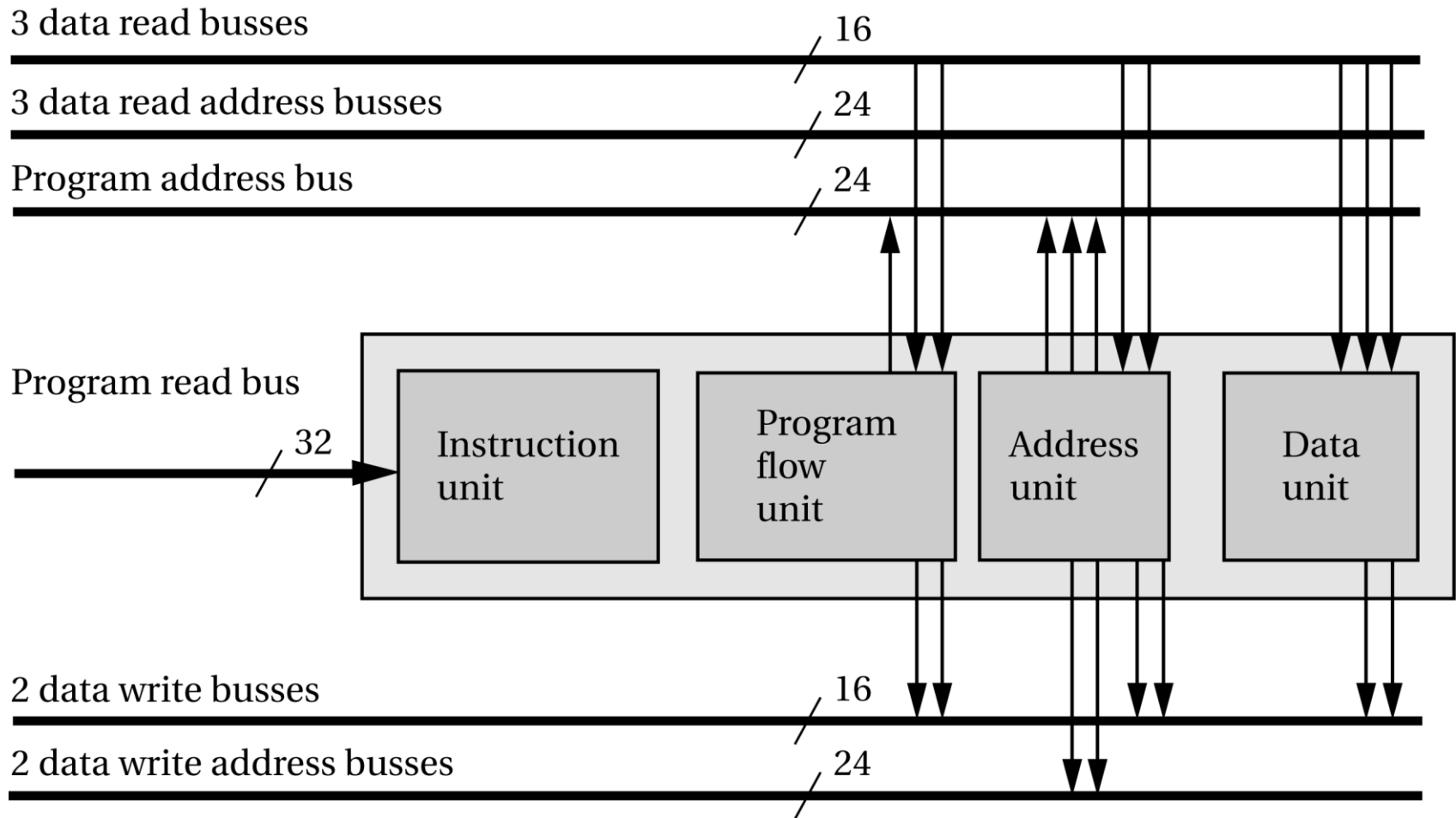## ARM

General Purpose:
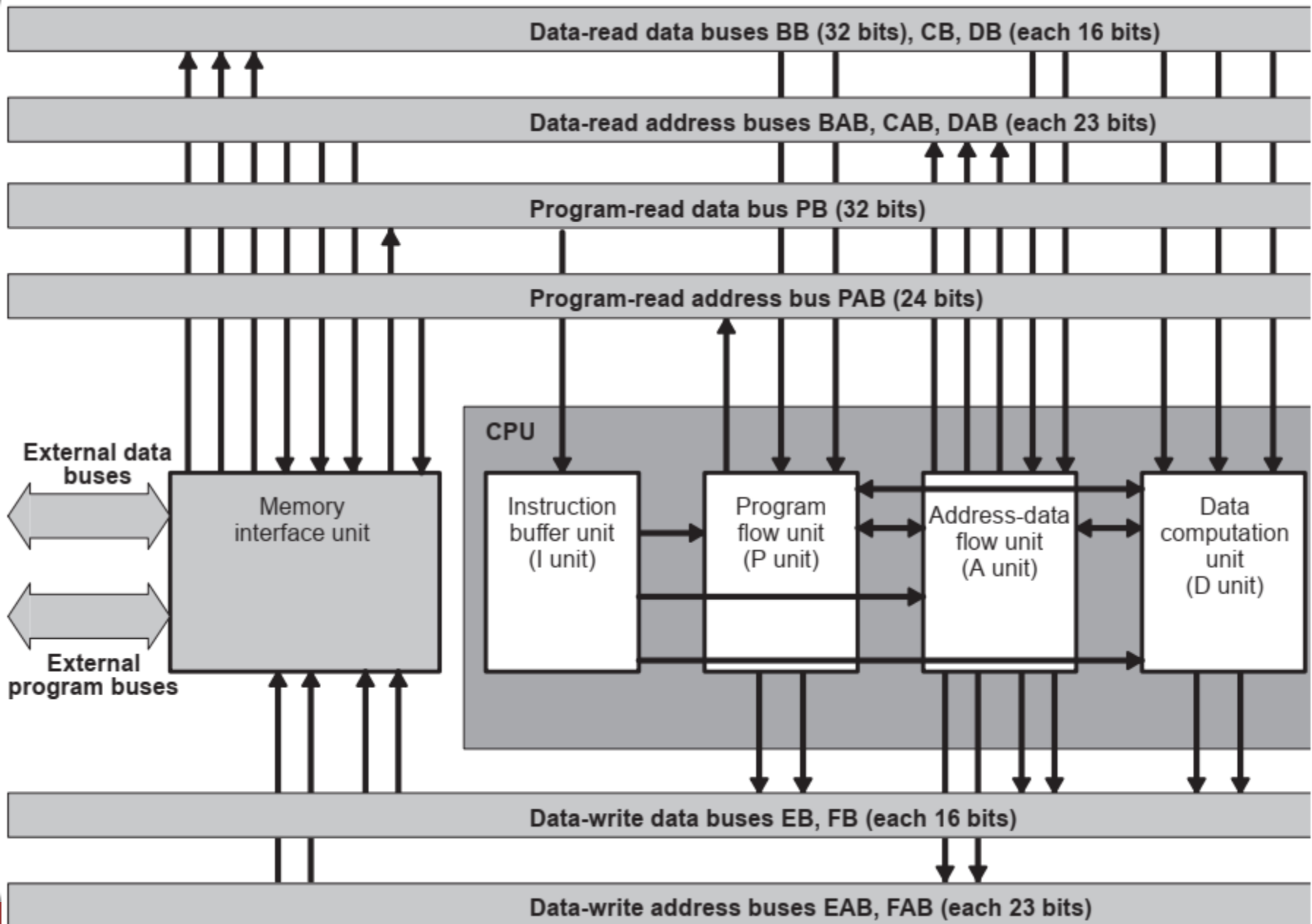R0 – R15 (32-bit)

SP: R13 (32-bit)
LR: R14 (32-bit)
PC: R15 (32-bit)

Status Register:
CPSR (32-bit)

3 data read busses — /16

3 data read address busses — /24

Program address bus — /24

Program read bus — /32 → Instruction unit | Program flow unit | Address unit | Data unit

2 data write busses — /16

2 data write address busses — /24

# TI C55x Microarchitecture

Data-read data buses BB (32 bits), CB, DB (each 16 bits)

Data-read address buses BAB, CAB, DAB (each 23 bits)

Program-read data bus PB (32 bits)

Program-read address bus PAB (24 bits)

**CPU**

**External data buses**

**External program buses**

Memory interface unit

Instruction buffer unit (I unit)

Program flow unit (P unit)

Address-data flow unit (A unit)

Data computation unit (D unit)

Data-write data buses EB, FB (each 16 bits)

Data-write address buses EAB, FAB (each 23 bits)

# TI C55x DSP (CISC)

## 3.1.1 Tips on Data Types

Give careful consideration to the data type size when writing your code. The C55x compiler defines a size for each C data type (signed and unsigned):

| | |
|---|---|
| char | 16 bits |
| short | 16 bits |
| int | 16 bits |
| long | 32 bits |
| long long | 40 bits |
| float | 32 bits |
| double | 64 bits |

Floating point values are in the IEEE format. Based on the size of each data type, follow these guidelines when writing your code:

❏ Avoid code that assumes that int and long types are the same size.

❏ Use the int data type for fixed-point arithmetic (especially multiplication) whenever possible. Using type long for multiplication operands will result in calls to a run-time library routine.

❏ Use int or unsigned int types rather than long for loop counters. The C55x has mechanisms for efficient hardware loops, but hardware loop counters are only 16 bits wide.

❏ Avoid code that assumes char is 8 bits or long long is 64 bits.

# TI C55x DSP (CISC)

```c
int sum(const int *a, int n)
{
    int total = 0;
    int i;
    for(i=0; i<n; i++)
    {
        total += a[i];
    }
    return total;
}
```

```
_sum:
;** Parameter deleted n == 9u
        MOV #0, T0 ; |3|
        RPT #9
            ADD *AR0+, T0, T0

        return    ; |11|
```

```c
void vecsum(const short *a, const short *b, short *c, unsigned int n)

{

    unsigned int i;


    for (i=0; i<=n-1; i++)

    {

        *c++ = *a++ + *b++;

    }

}
```

```
_vecsum:
        SUB #1, T0, AR3

        MOV AR3, BRC0

        RPTBLOCAL L2-1

            ADD *AR0+, *AR1+, AC0 ; |7|

            MOV HI(AC0), *AR2+ ; |7|

L2:

        return
```

# 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

| memory mapped registers |
|---|
| main data page 1 |
| main data page 2 |
| ... |
| main data page 127 |
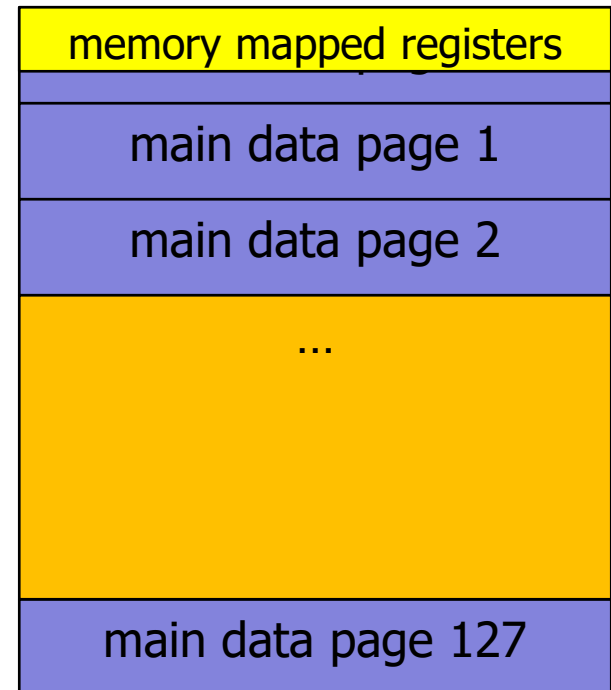
# C55x Addressing Modes

- 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.
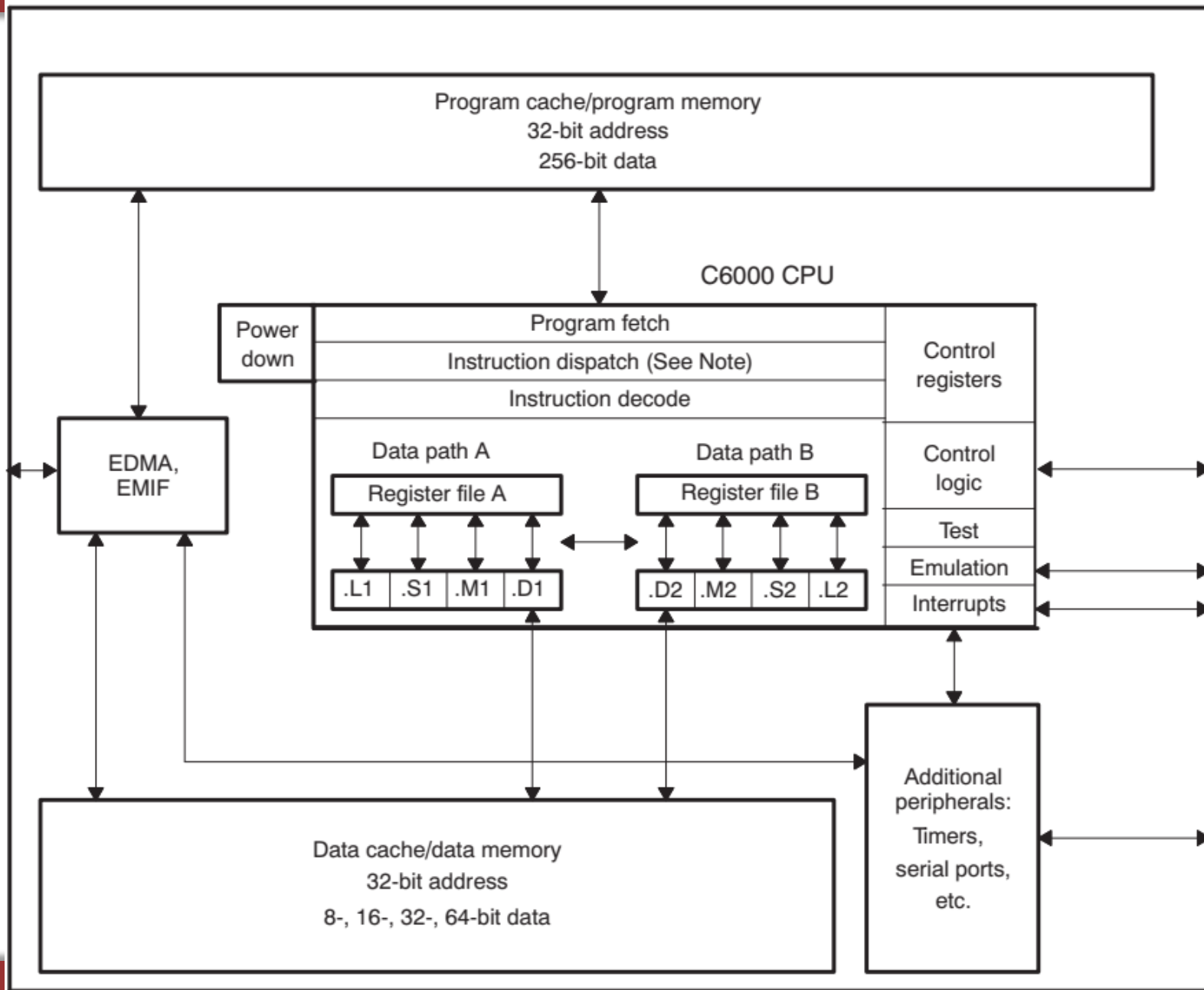  - Slow return puts return address/loop on stack.

# C64x DSP (VLIW)

- Can execute up to 8 32-bit instructions at a time:
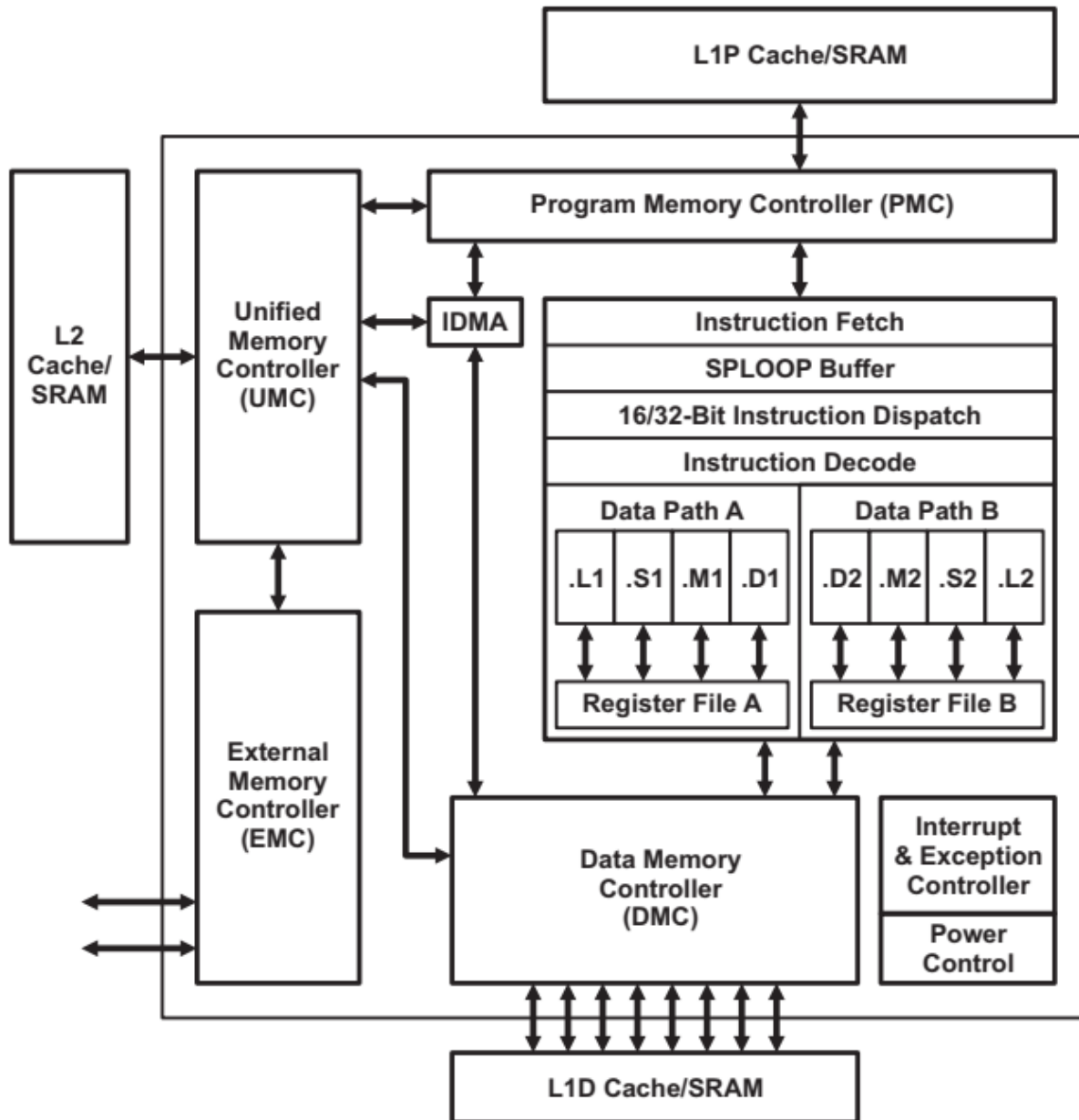
- Has a RISC like structure

**Figure 3-1. Basic Format of a Fetch Packet**

| 31 0 | 31 0 | 31 0 | 31 0 | 31 0 | 31 0 | 31 0 | 31 0 |
|---|---|---|---|---|---|---|---|
| p | p | p | p | p | p | p | p |
| Instruction A | Instruction B | Instruction C | Instruction D | Instruction E | Instruction F | Instruction G | Instruction H |
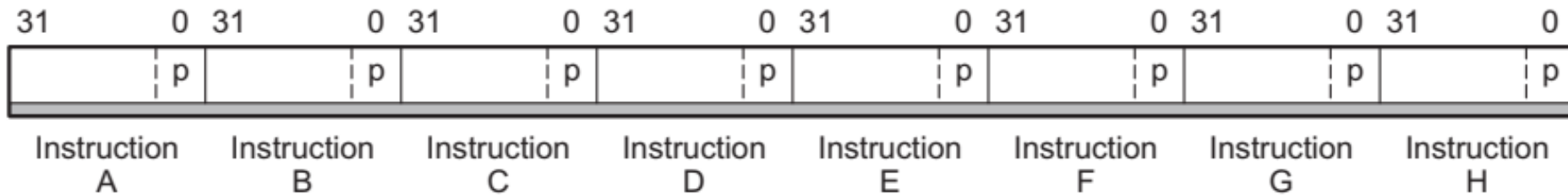
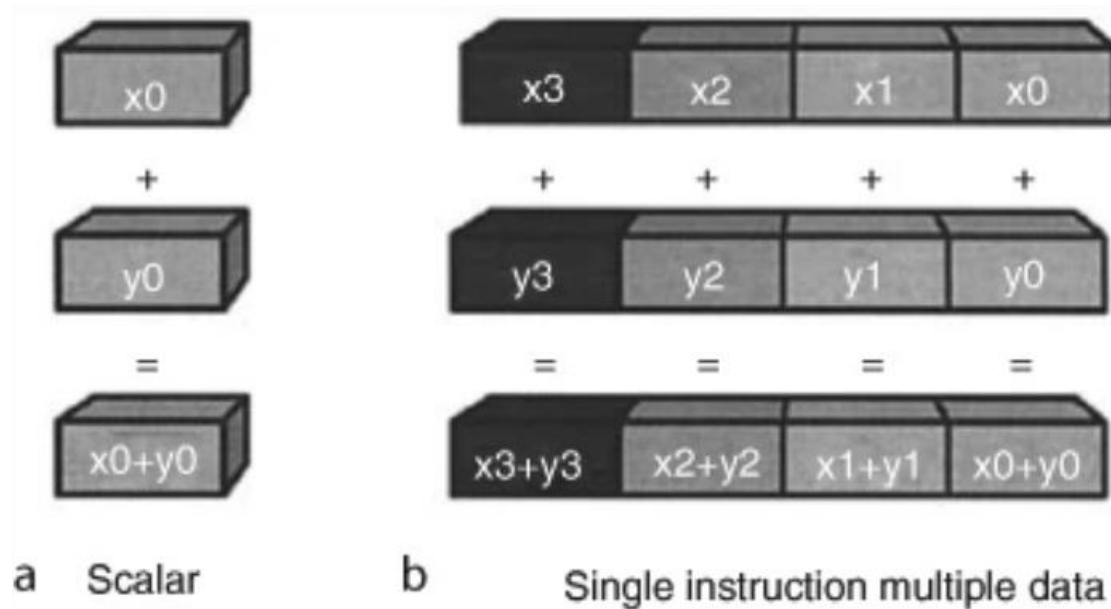# TI C64x Block Diagram

# 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

# Flynn's Taxonomy
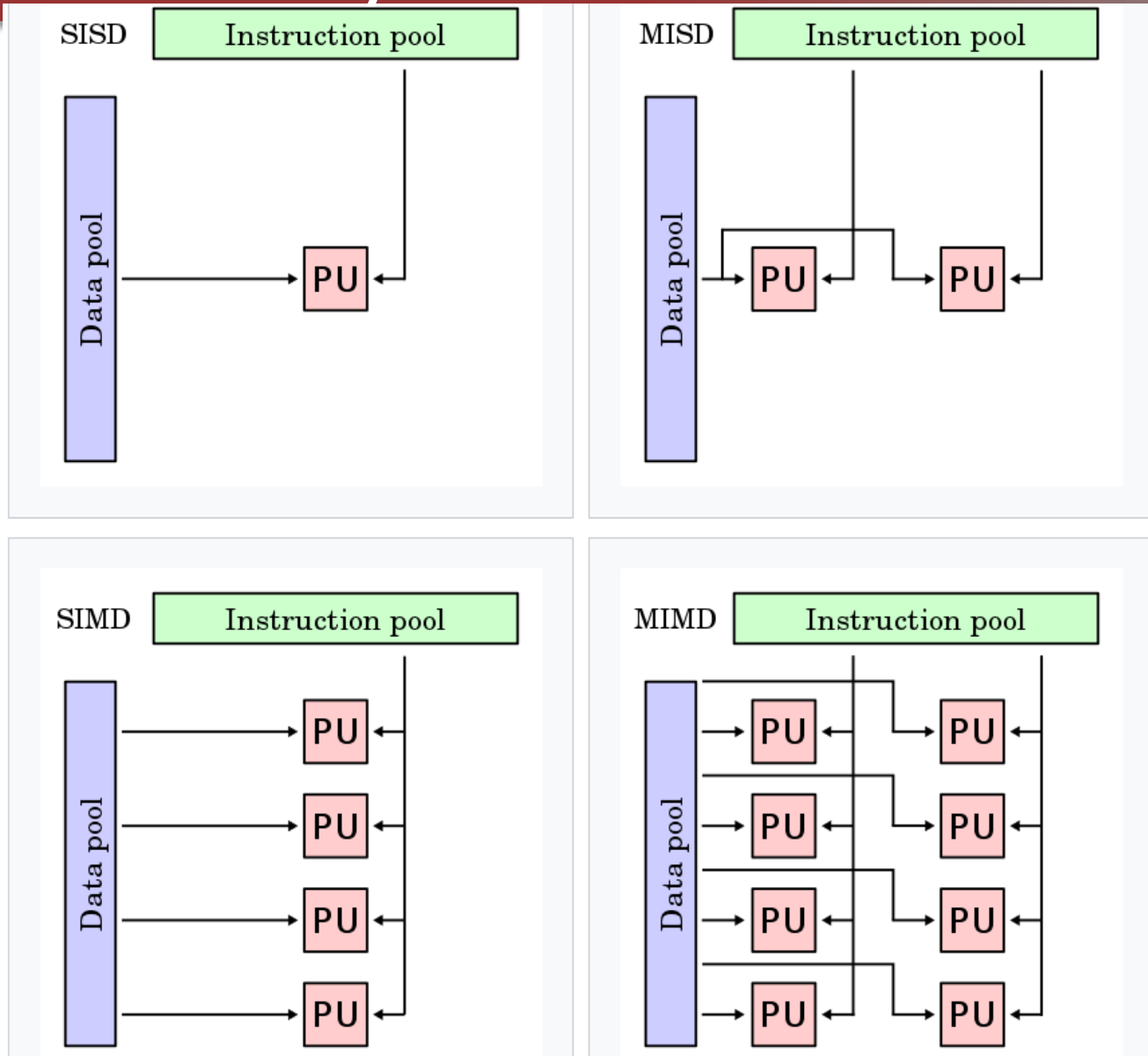


Link: https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

# SIMD: Example



a    Scalar

b    Single instruction multiple data

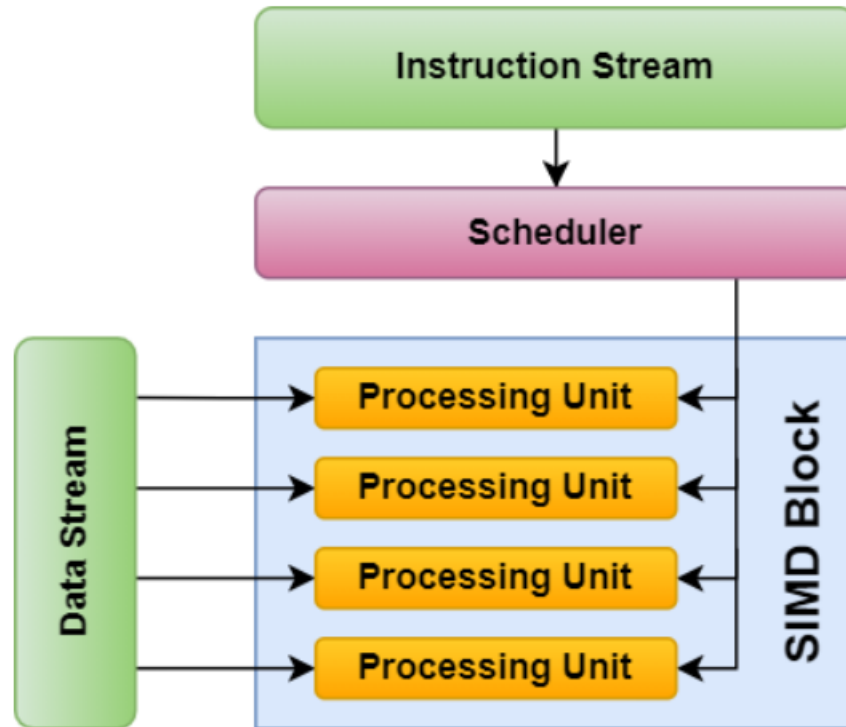Link: https://link.springer.com/referenceworkentry/10.1007/978-0-387-78414-4_220/figures/1_220

# SIMD: Example



Illustration of a 4-lane SIMD block.

https://www.rastergrid.com/blog/gpu-tech/2022/02/simd-in-the-gpu-world/