

# CprE 488 – Embedded Systems Design

## Lecture 3 – Processors and Memory

Phillip Jones

Electrical and Computer Engineering

Iowa State University

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

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

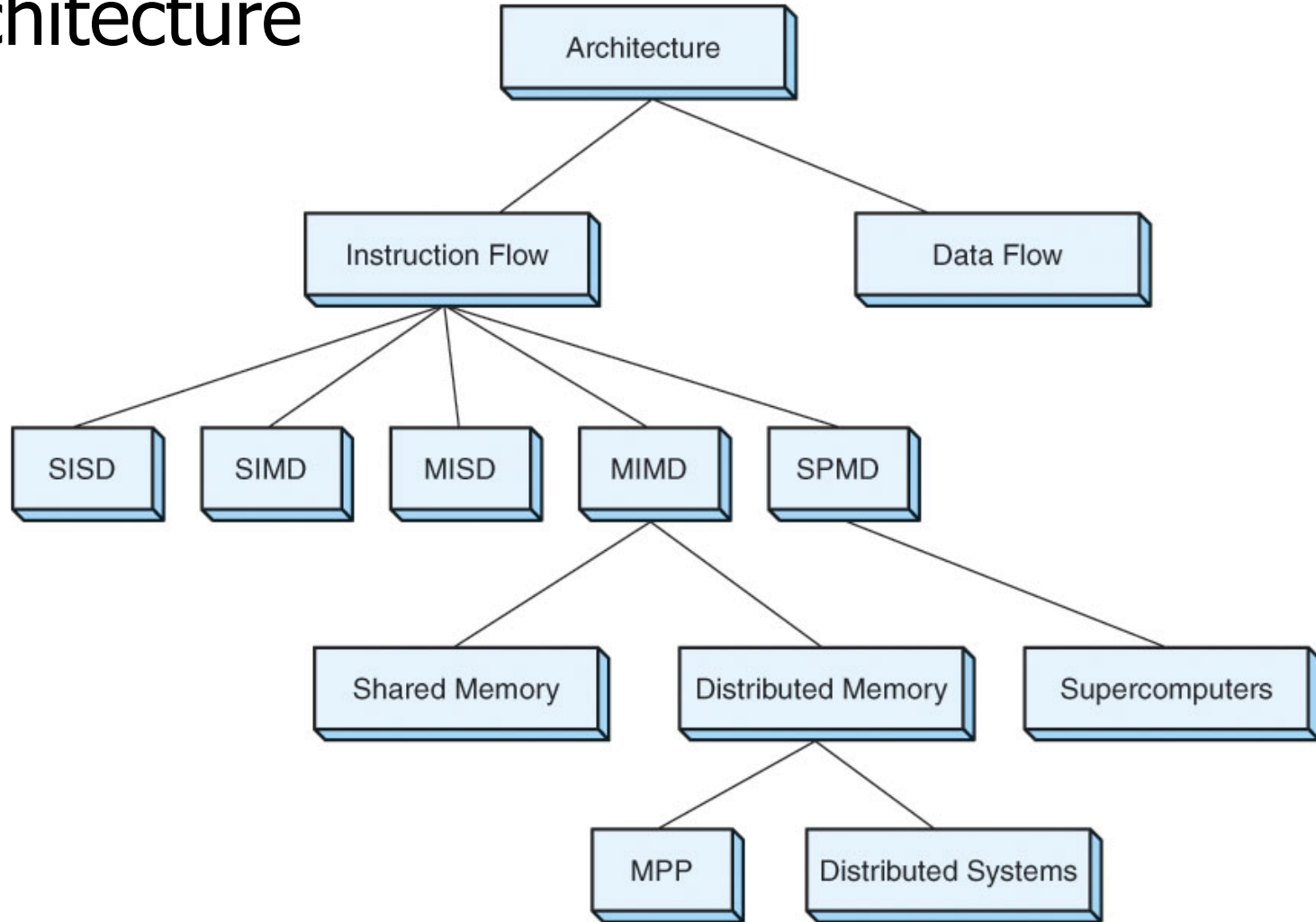
*Although computer memory is no longer expensive, there's always a finite size buffer somewhere. – Benoit Mandelbrot*

# This Week's Topic

- Embedded processor design tradeoffs
- ISA and programming models
- Memory system mechanics
- Case studies:
  - ARM v7 CPU (RISC)
  - TI C55x DSP (CISC)
  - TI C64x DSP (VLIW)
- Reading: Wolf –
  - Chapter 2 (Instruction Sets),
  - Chapter 3.5 (Memory System Mechanisms)

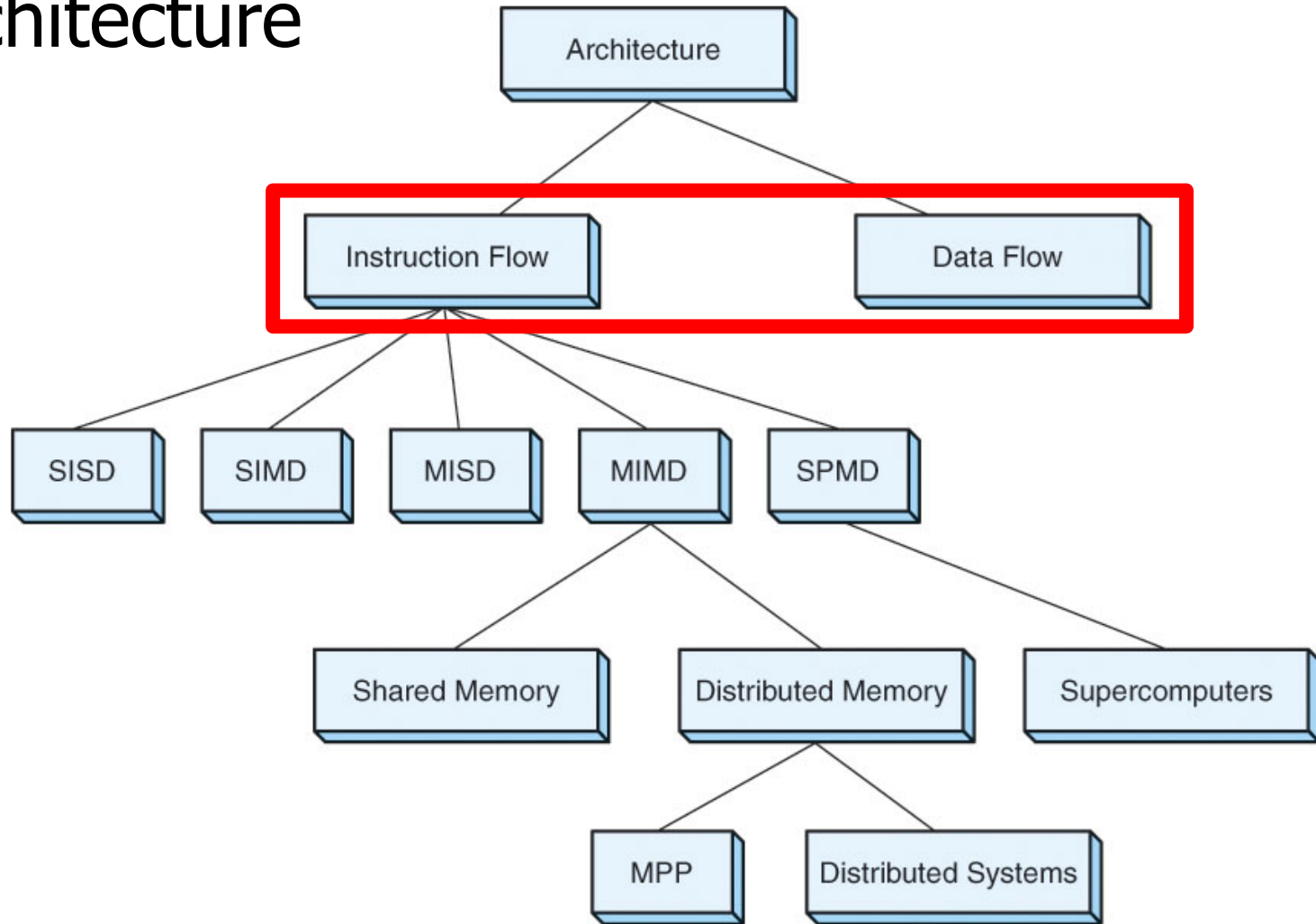
# Flynn's (Updated) Taxonomy

- AKA the “alphabet soup” of computer architecture



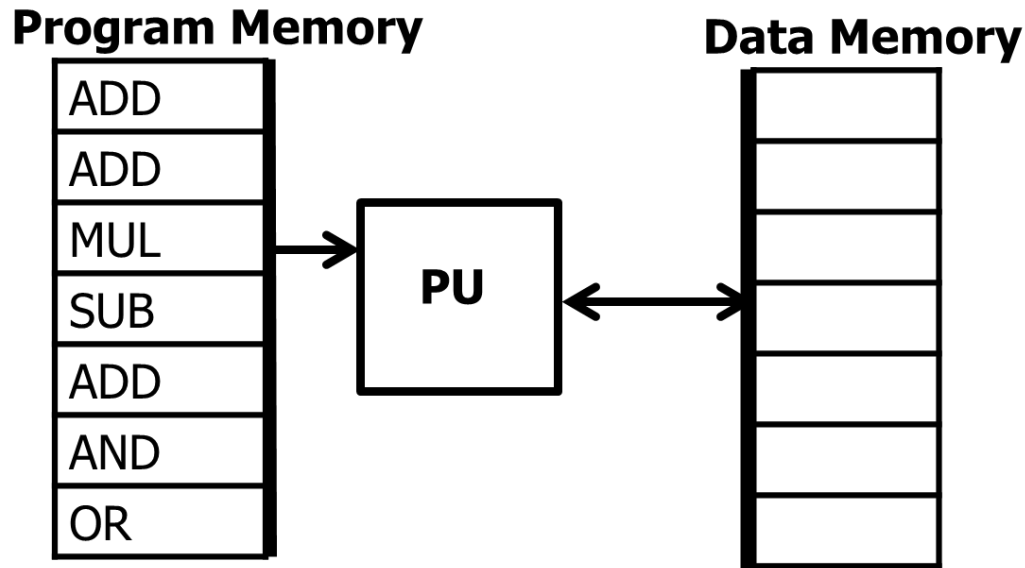
# Flynn's (Updated) Taxonomy

- AKA the “alphabet soup” of computer architecture

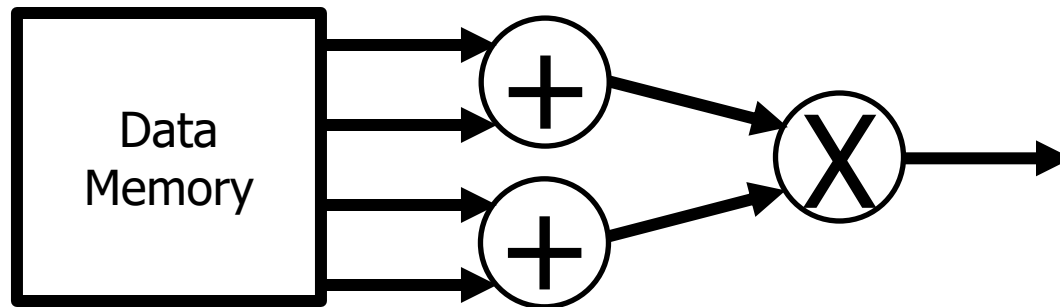


# Flynn's Taxonomy: Instruction vs Data Flow

## Instruction Flow

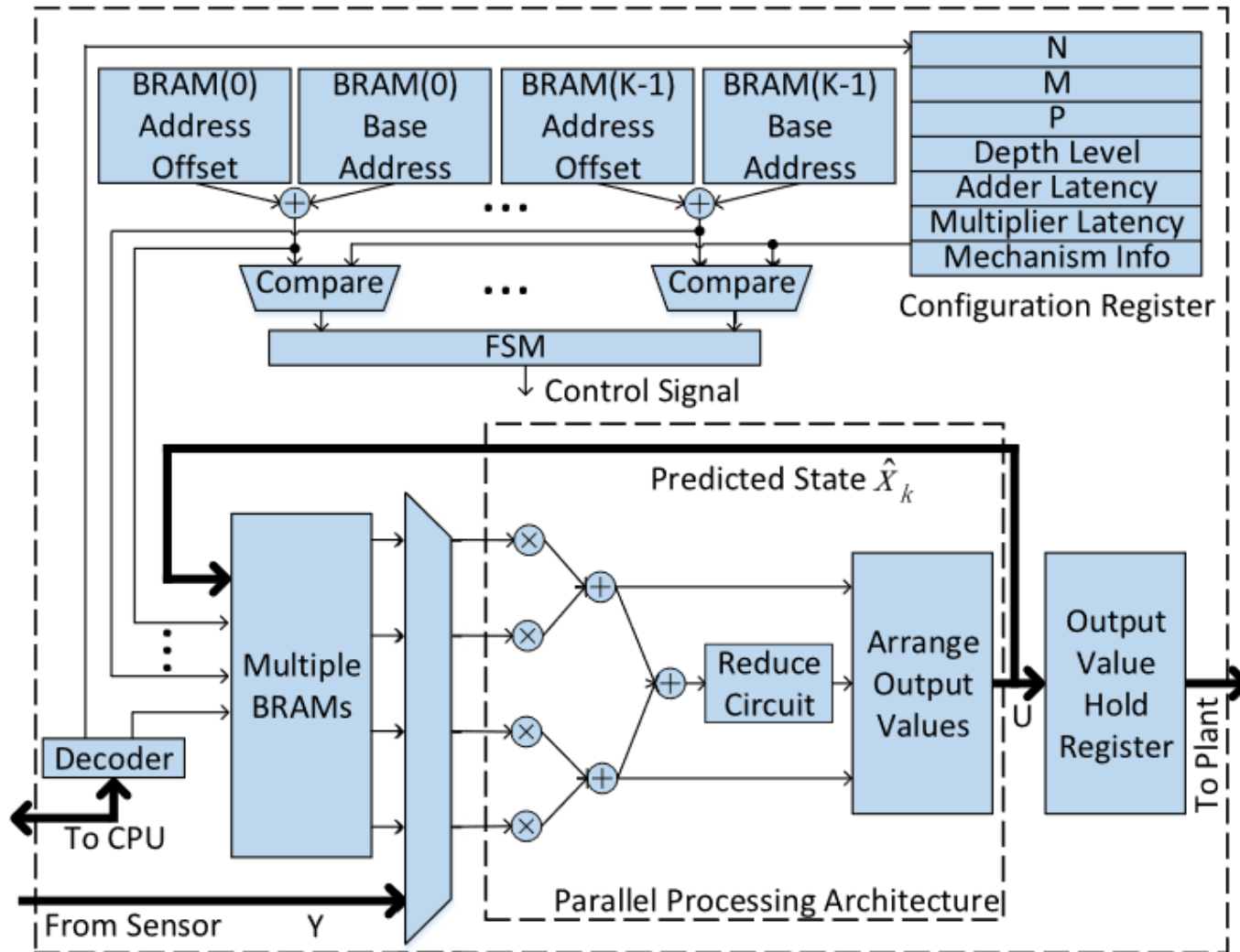


## Data Flow



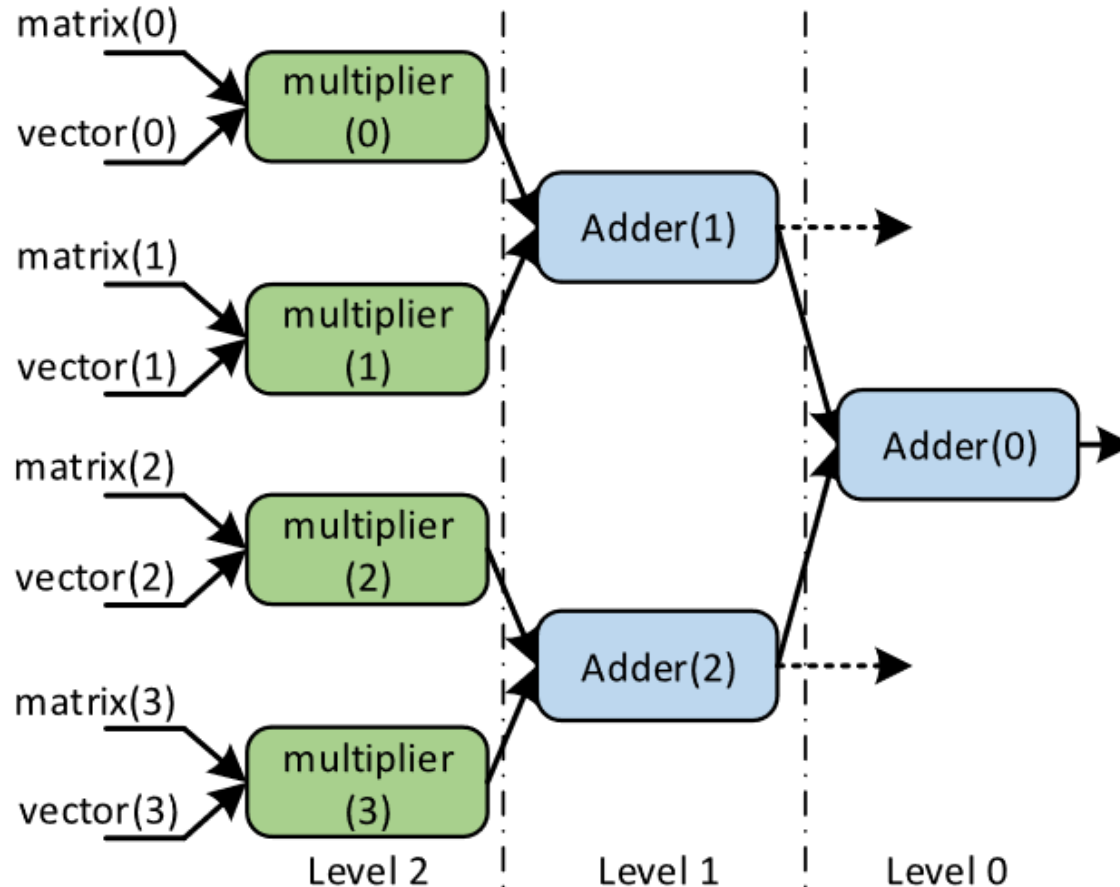
# Flynn's Taxonomy: Data Flow

## Example: Custom Hardware for LQR Controller



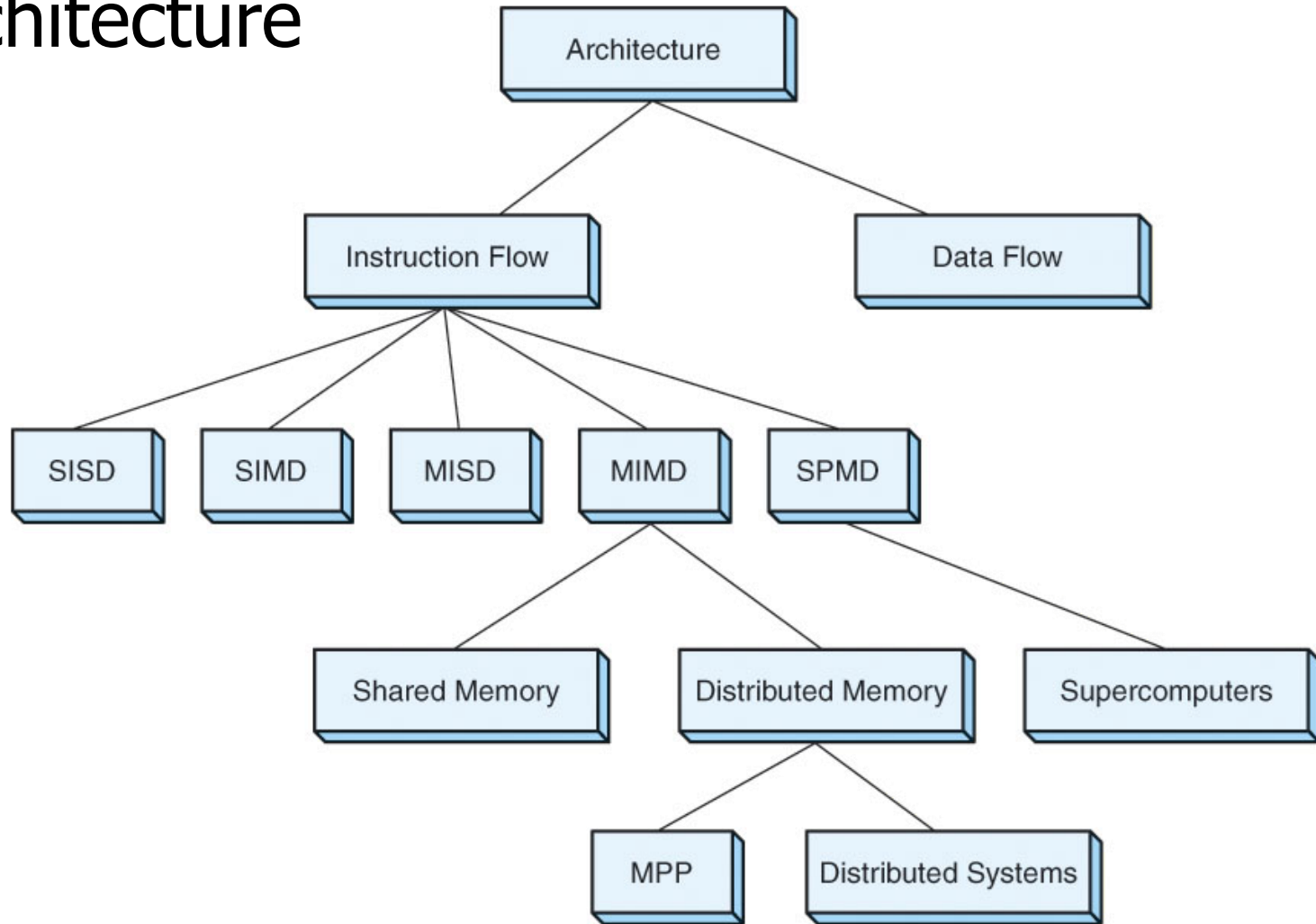
# Flynn's Taxonomy: Data Flow

Vector Matrix multiplication, primary computation of LQR controller



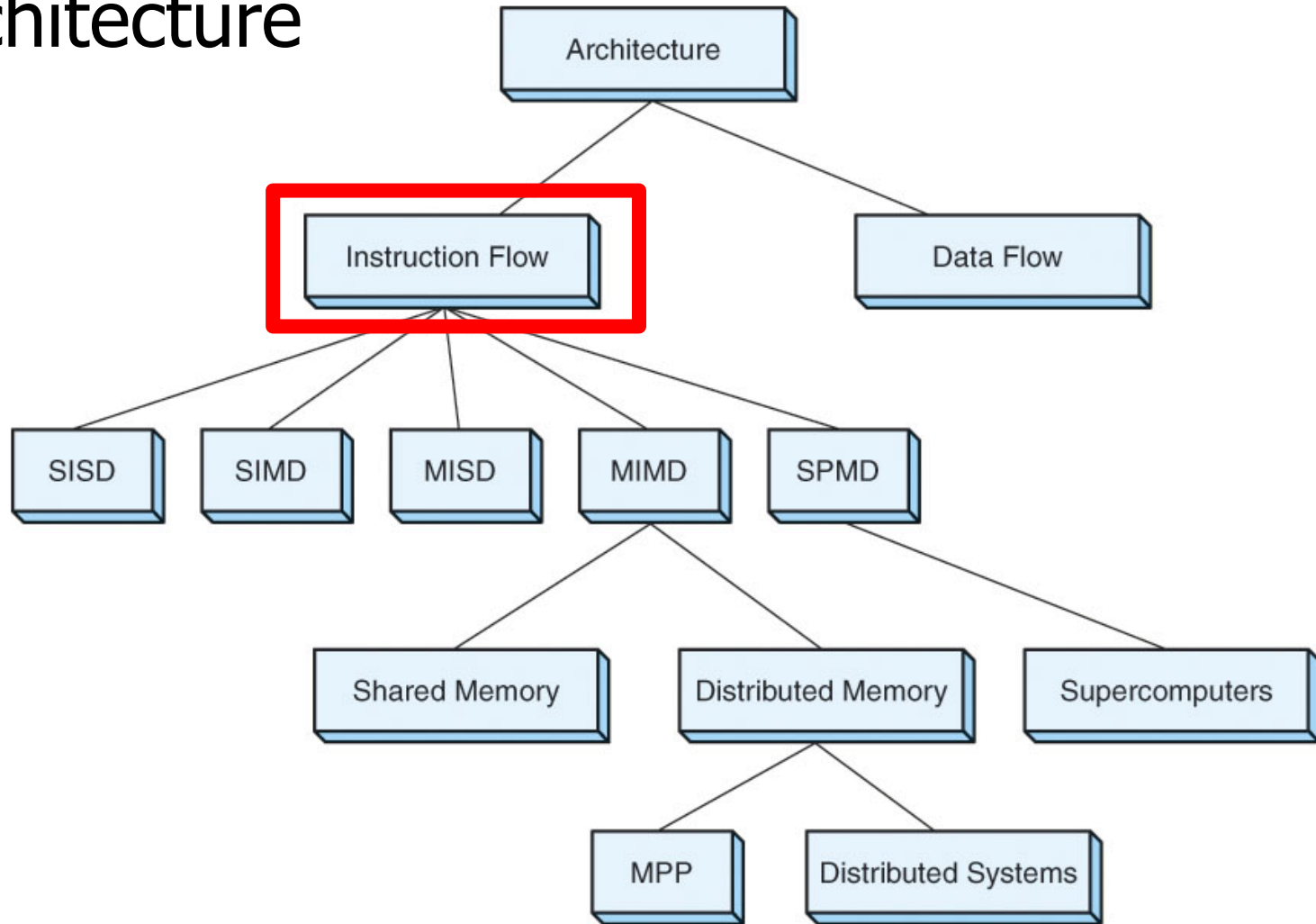
# Flynn's (Updated) Taxonomy

- AKA the “alphabet soup” of computer architecture



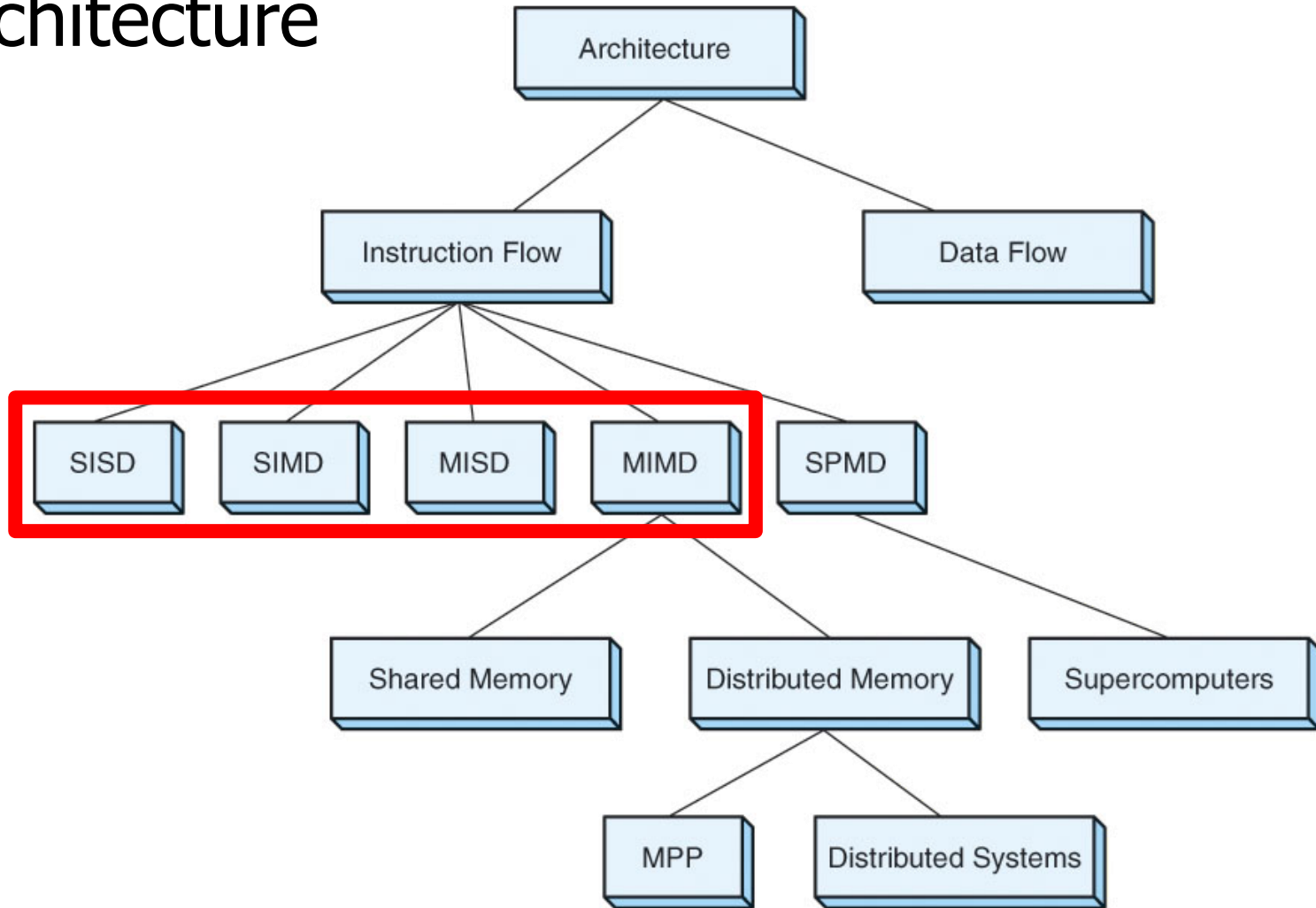
# Flynn's (Updated) Taxonomy

- AKA the “alphabet soup” of computer architecture

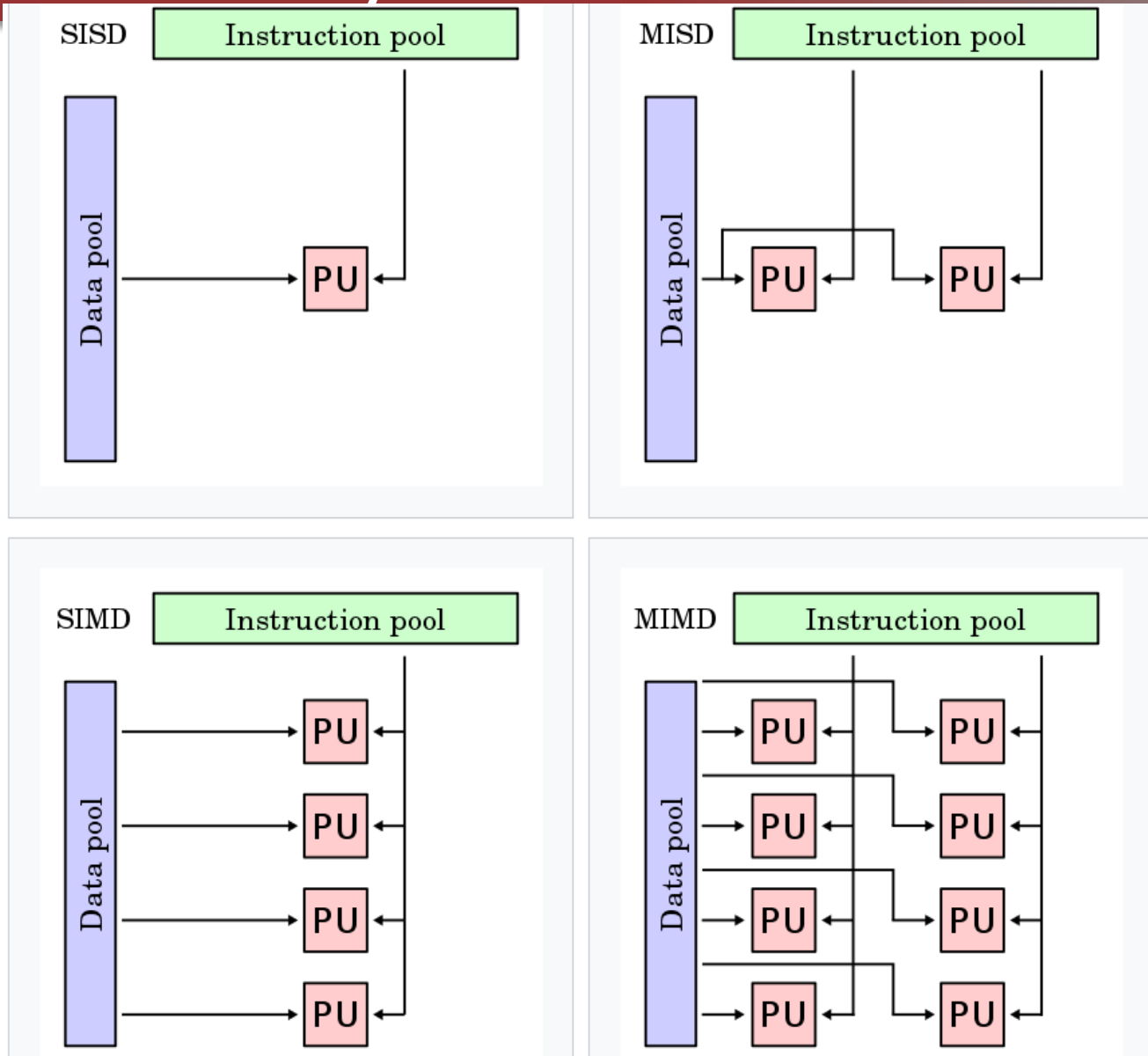


# Flynn's (Updated) Taxonomy

- AKA the "alphabet soup" of computer architecture

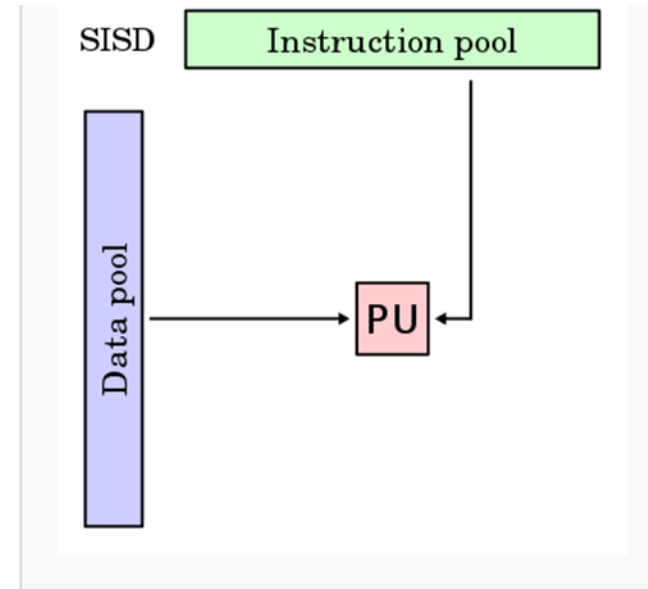


# Flynn's Taxonomy: Instruction Flow

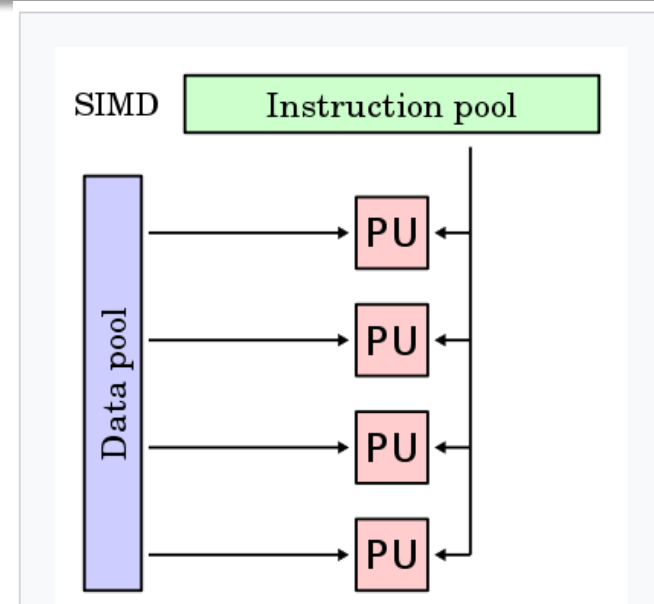


Link: [https://en.wikipedia.org/wiki/Flynn%27s\\_taxonomy](https://en.wikipedia.org/wiki/Flynn%27s_taxonomy)

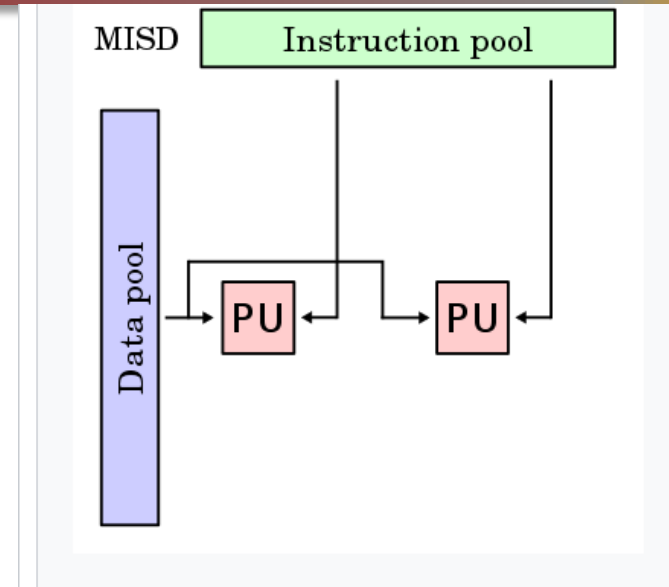
# Single Instruction Singel Data (SISD)



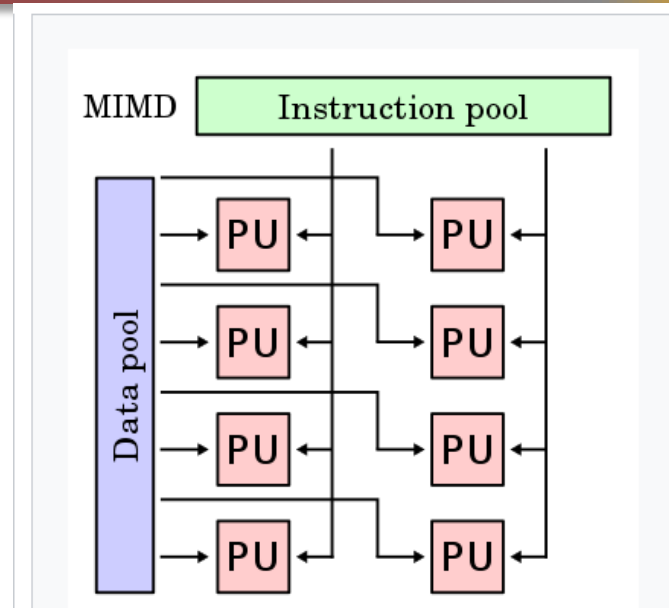
# Single Instruction Multiple Data (SIMD)



# Multiple Instruction Single Data (MISD)



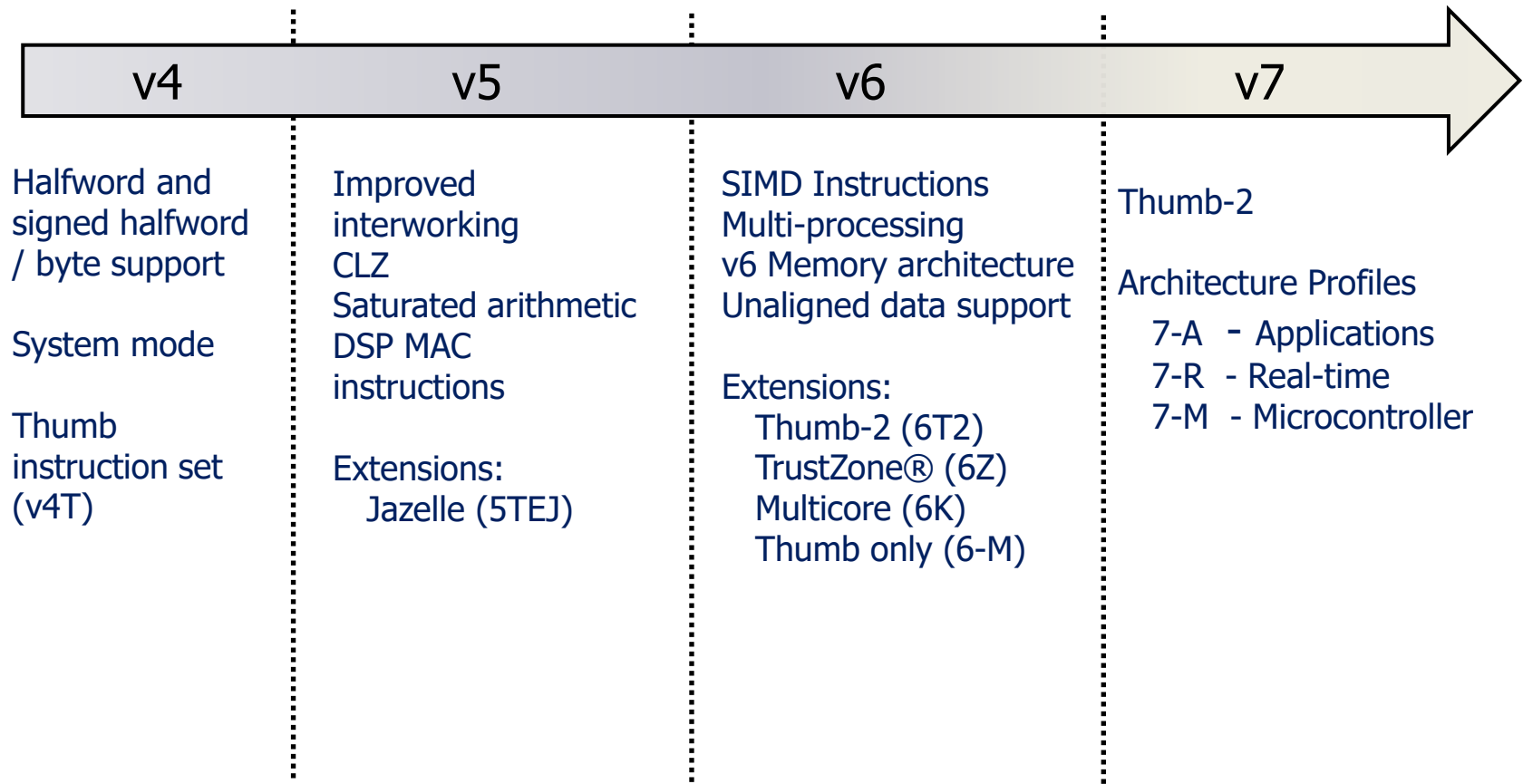
# Multiple Instruction Multiple Data (MIMD)



# Case Studies

- **ARM** v7 CPU (RISC)
- TI C55x DSP (CISC)
- TI C64x DSP (VLIW)

# ARM Architecture Revisions

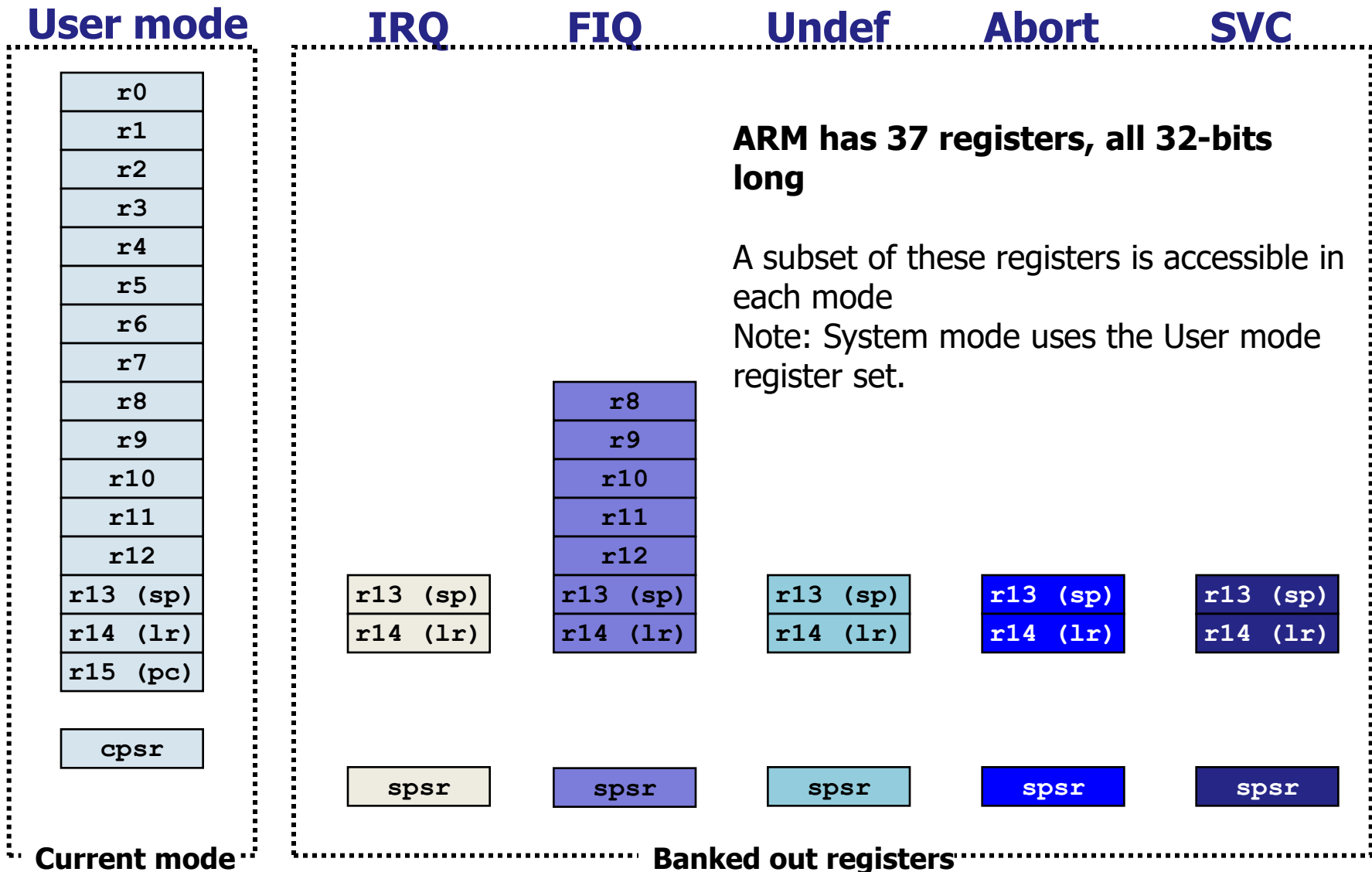


- Note that implementations of the same architecture can be different
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

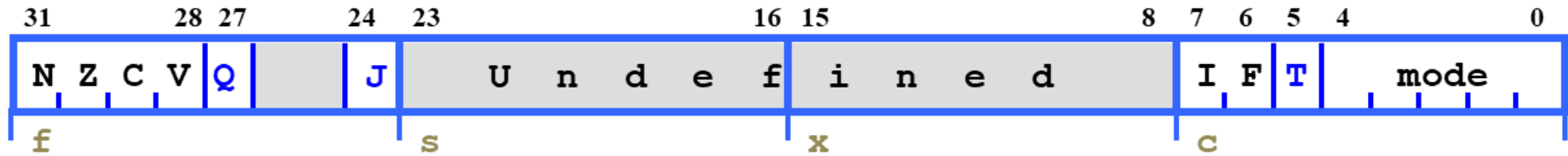
# Data Sizes and Instruction Sets

- 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

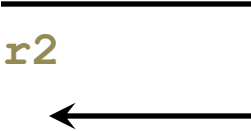


- Condition code flags
  - N = **N**egative result from ALU
  - Z = **Z**ero result from ALU
  - C = ALU operation **C**arried out
  - V = ALU operation o**V**erflowed
- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred
- J bit
  - Architecture 5TEJ only
  - 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
```

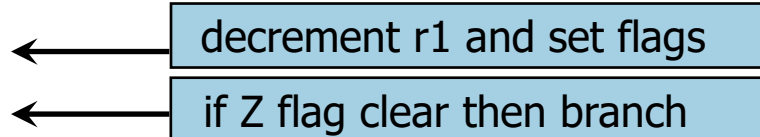
A diagram showing a branch from the BEQ instruction to the skip label. A horizontal line extends from the right side of the BEQ instruction, then turns down and then left, ending in an arrowhead pointing to the skip label.

```
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
```

Two blue boxes with black text and arrows pointing to the instructions. The top box contains the text "decrement r1 and set flags" and has an arrow pointing to the SUBS instruction. The bottom box contains the text "if Z flag clear then branch" and has an arrow pointing to the BNE instruction.

decrement r1 and set flags

if Z flag clear then branch

# Condition Codes

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

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

### conditional

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

- 5 instructions
- 5 words
- 5 or 6 cycles

- 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:

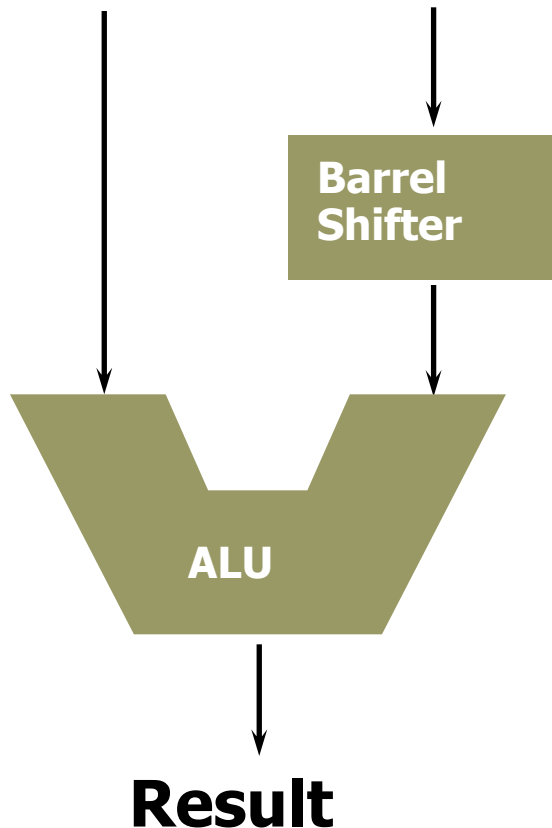
**<Operation>{<cond>} {S} Rd, Rn, Operand2**

- 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**



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?

```
MOVN    r3, #0
```

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

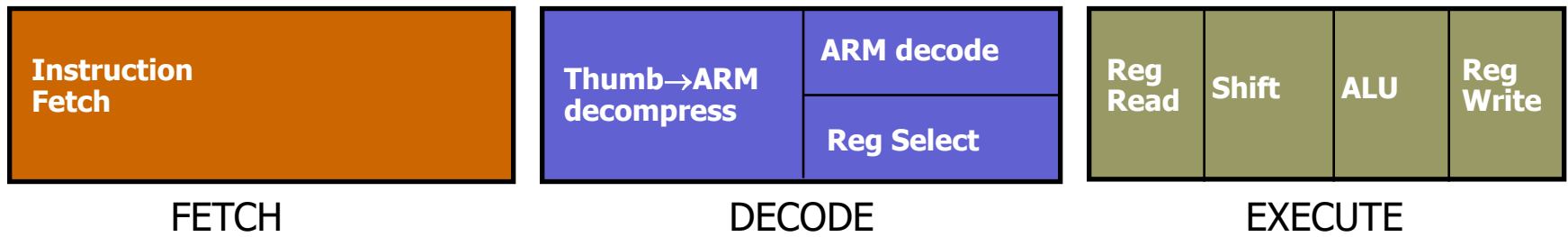
```
MOVS    r7, r7    ; set the flags  
RSBMI   r7, r7, #0 ; if neg, r7=0-r7
```

3. Multiply a number by 35, guaranteeing that it executes in 2 core clock cycles (i.e. in two instructions)

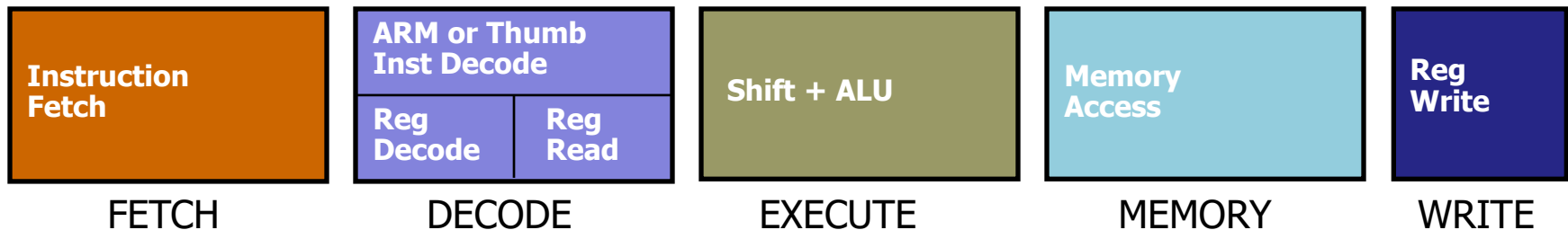
```
ADD     r9, r8, r8, LSL #2    ; r9=r8*5  
RSB    r10, r9, r9, LSL #3    ; r10=r9*7
```

# ARM Pipeline Evolution

## ARM7TDMI

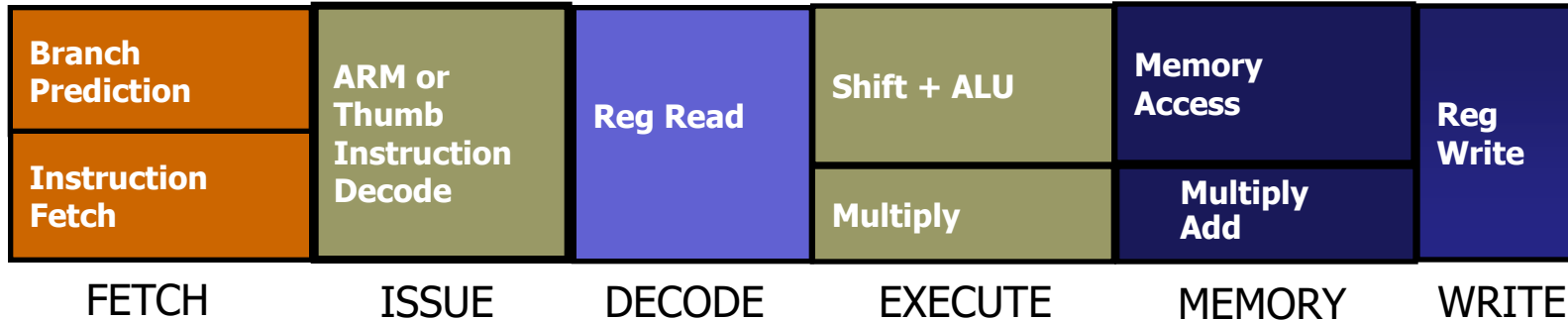


## ARM9TDMI

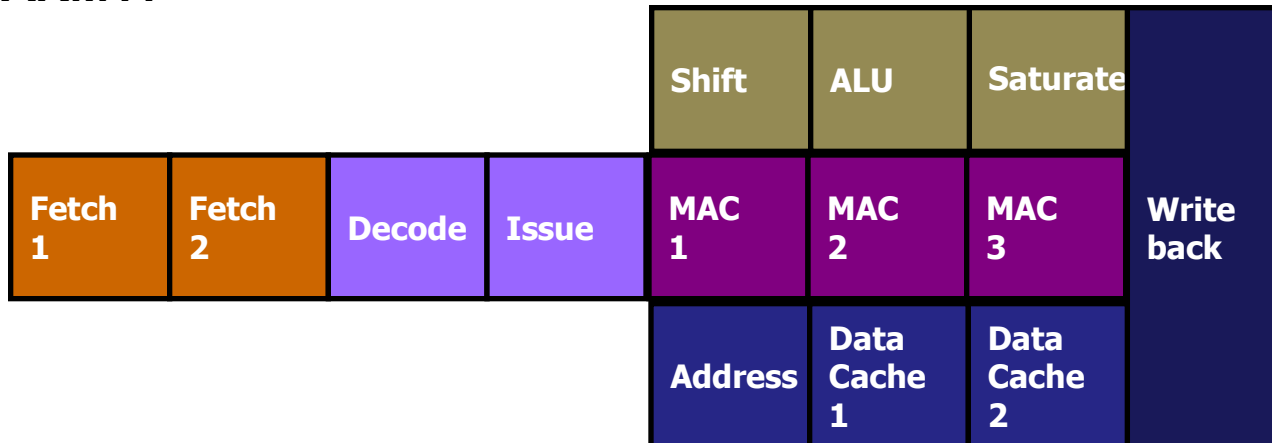


# ARM Pipeline Evolution (cont.)

## ARM10



## ARM11

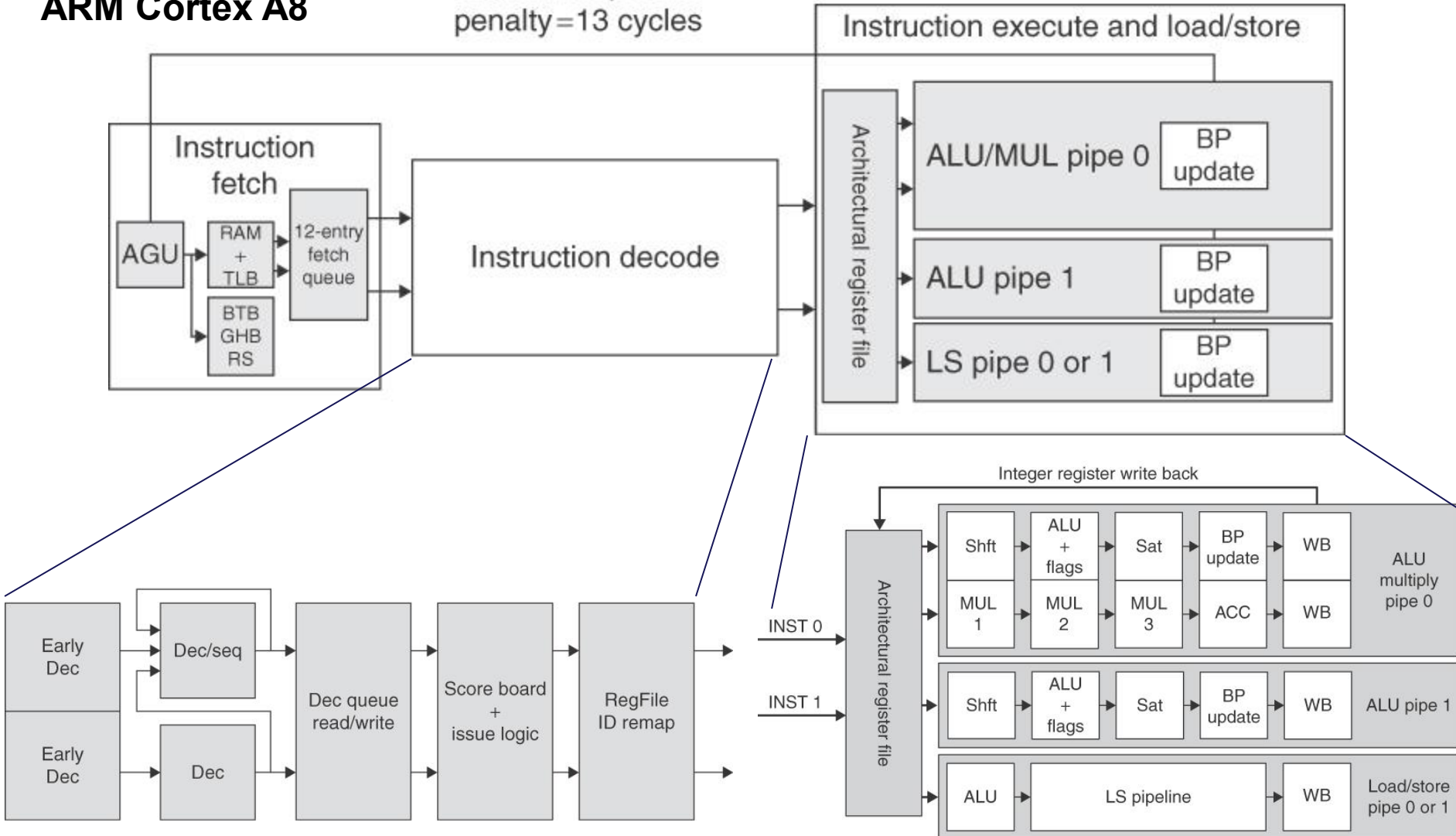


# ARM Pipeline Evolution (cont.)

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

## ARM Cortex A8

Branch mispredict penalty = 13 cycles



# Case Studies

- ARM v7 CPU (RISC)
- TI C55x DSP (CISC)
- TI C64x DSP (VLIW)

# Digital Signal Processor (DSP)

- 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)

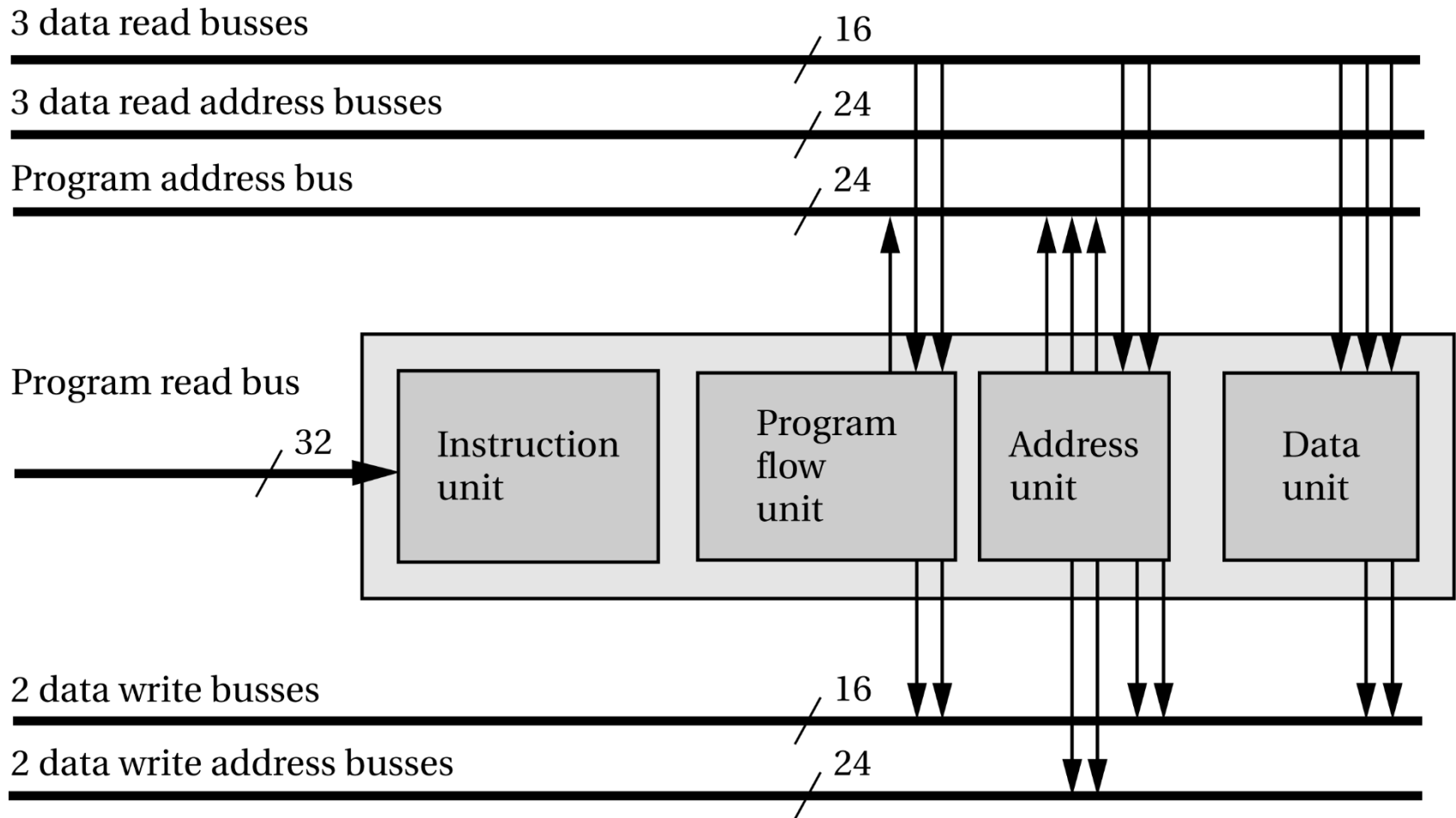
SP: R13 (32-bit)

LR: R14 (32-bit)

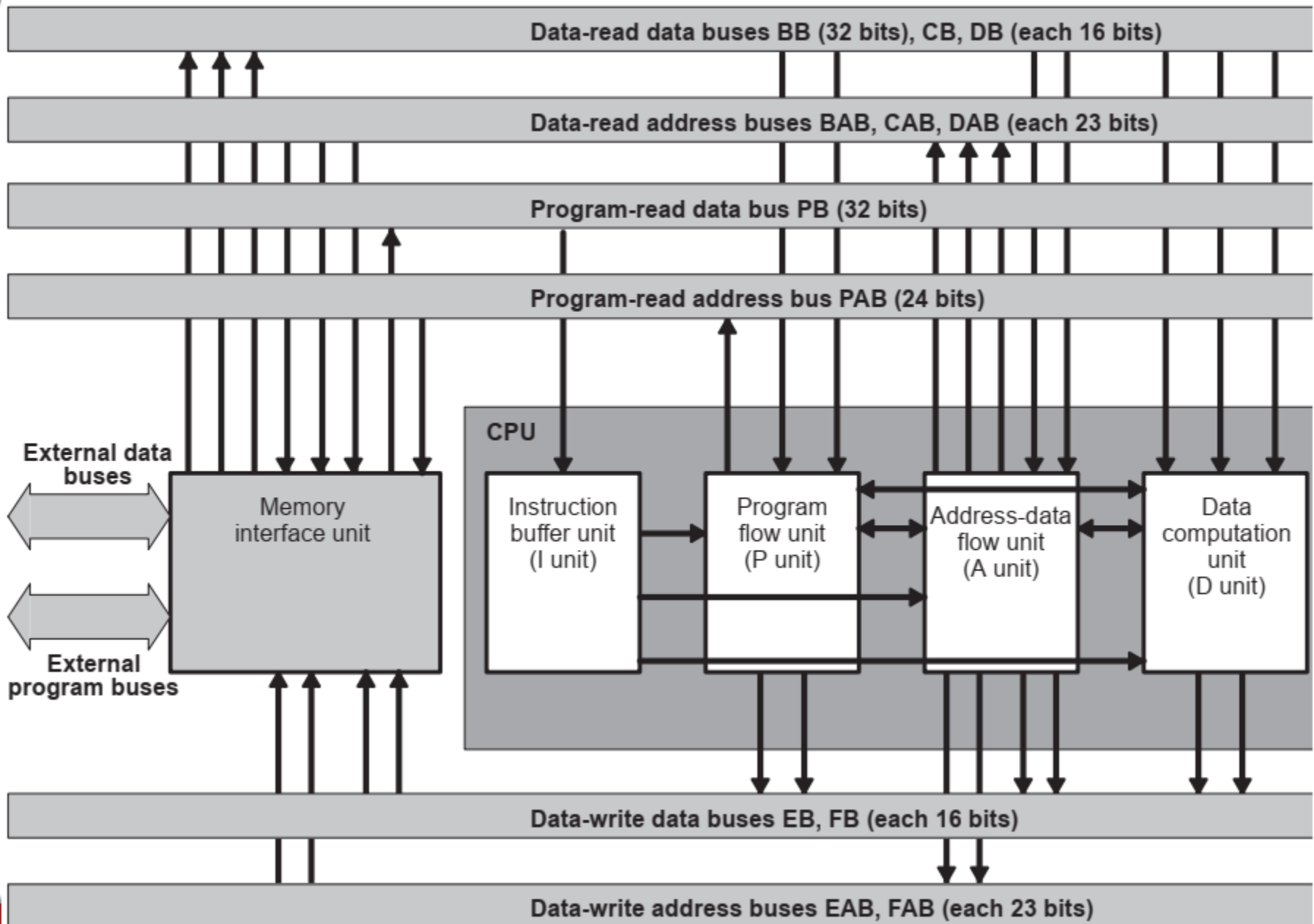
PC: R15 (32-bit)

Status Register:  
CPSR (32-bit)

# TI C55x Microarchitecture



# TI C55x Microarchitecture



# 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):

<code>char</code>	16 bits
<code>short</code>	16 bits
<code>int</code>	16 bits
<code>long</code>	32 bits
<code>long long</code>	40 bits
<code>float</code>	32 bits
<code>double</code>	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)

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

## ARM:

```
MOVW R4, #9 ; Loop count
          ; R0 as location of a[]
          ; R2 as total

loop:
LDR R1, [R0], #4 ;get int from a[i]
ADD R2, R1      ; total += a[i]
SUB R4, #1      ; i--
CMP R4, #0
BNE loop        ; exit when i = 0
STR R2, [R5]; ;store total
```

## C55x:

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

return ; |11|
```

# TI C55x DSP (CISC)

```
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
```

## 4.2.1 Built-In Parallelism

Instructions that have built-in parallelism perform two different operations in parallel. In the algebraic syntax, they can be identified by the comma that separates the two operations, as in the following example:

```
AC0 = *AR0 * coef(*CDP),      ; The data referenced by AR0 is multiplied by
AC1 = *AR1 * coef(*CDP)      ; a coefficient referenced by CDP. At the same time
                               ; the data referenced by AR1 is multiplied by the
                               ; same coefficient.
```

In the mnemonic syntax, they can be identified by a double colon (::) that separates the two operations. The preceding example in the mnemonic syntax is:

```
MPY *AR0, *CDP, AC0          ; The data referenced by AR0 is multiplied by
:: MPY *AR1, *CDP, AC1      ; a coefficient referenced by CDP. At the same time
                               ; the data referenced by AR1 is multiplied by the
                               ; same coefficient.
```

## 4.2.2 User-Defined Parallelism

Two instructions may be placed in parallel to have them both execute in a single cycle. The two instructions are separated by the || separator. One of the two instructions may have built-in parallelism. The following algebraic code example shows a user-defined parallel instruction pair. One of the instructions in the pair also features built-in parallelism.

```
AC0 = AC0 + (*AR3+ * coef(*CDP+)),      ; 1st instruction (has built-in parallelism)
AC1 = AC1 + (*AR4+ * coef(*CDP+))      ; 2nd instruction
|| repeat(CSR)
```

The equivalent mnemonic code example is:

```
MPY *AR3+, *CDP+, AC0          ; 1st instruction (has built-in parallelism)
:: MPY *AR4+, *CDP+, AC1      ; 2nd instruction
|| RPT CSR
```

# Case Studies

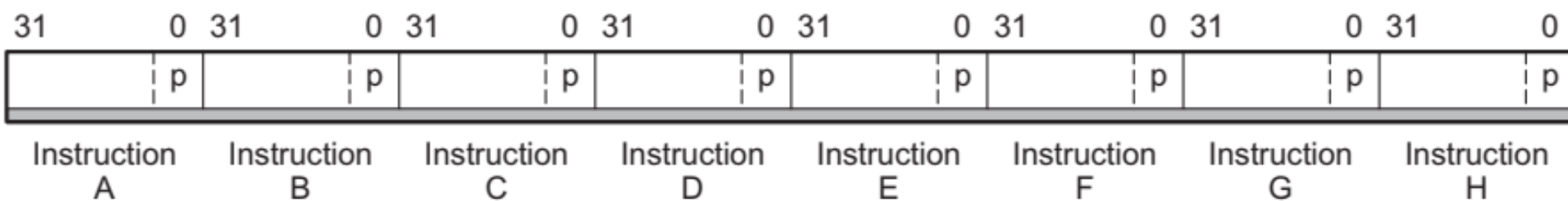
- ARM v7 CPU (RISC)
- TI C55x DSP (CISC)
- **TI C64x DSP (VLIW)**

# C64x DSP (VLIW Computing Architecture)

# C64x DSP (VLIW Computing Architecture)

- **VLIW**: **V**ery **L**ong **I**nstruction **W**ord Computing Architecture
- Can execute up to 8 32-bit instructions in parallel (256-bit inst.)
- Has a RISC-like structure

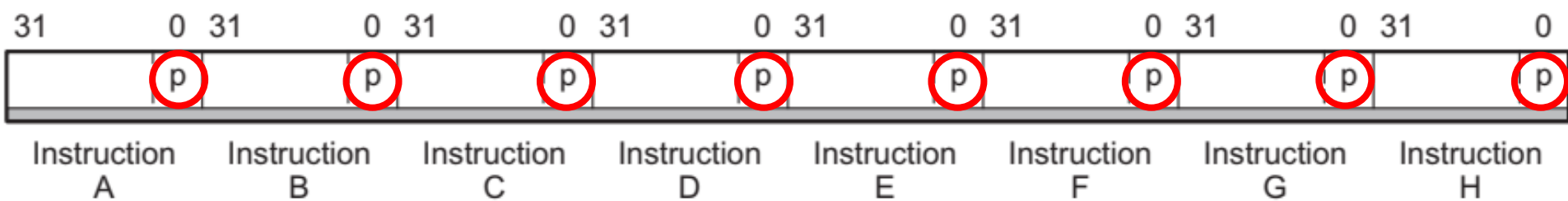
**Figure 3-1. Basic Format of a Fetch Packet**



# C64x DSP (VLIW Computing Architecture)

- **VLIW**: **V**ery **L**ong **I**nstruction **W**ord Computing Architecture
- Can execute up to 8 32-bit instructions in parallel (256-bit inst.)
- Has a RISC-like structure

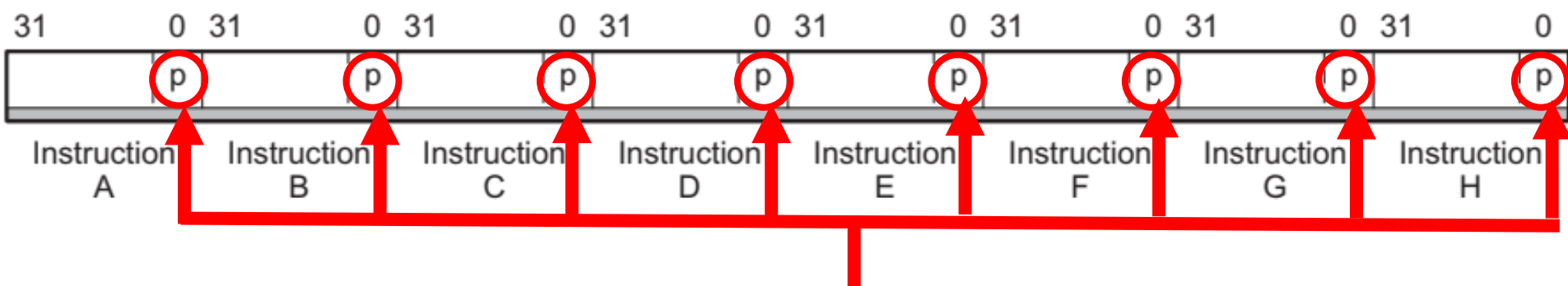
Figure 3-1. Basic Format of a Fetch Packet



# C64x DSP (VLIW Computing Architecture)

- **VLIW**: **V**ery **L**ong **I**nstruction **W**ord Computing Architecture
- Can execute up to 8 32-bit instructions in parallel (256-bit inst.)
- Has a RISC-like structure

Figure 3-1. Basic Format of a Fetch Packet

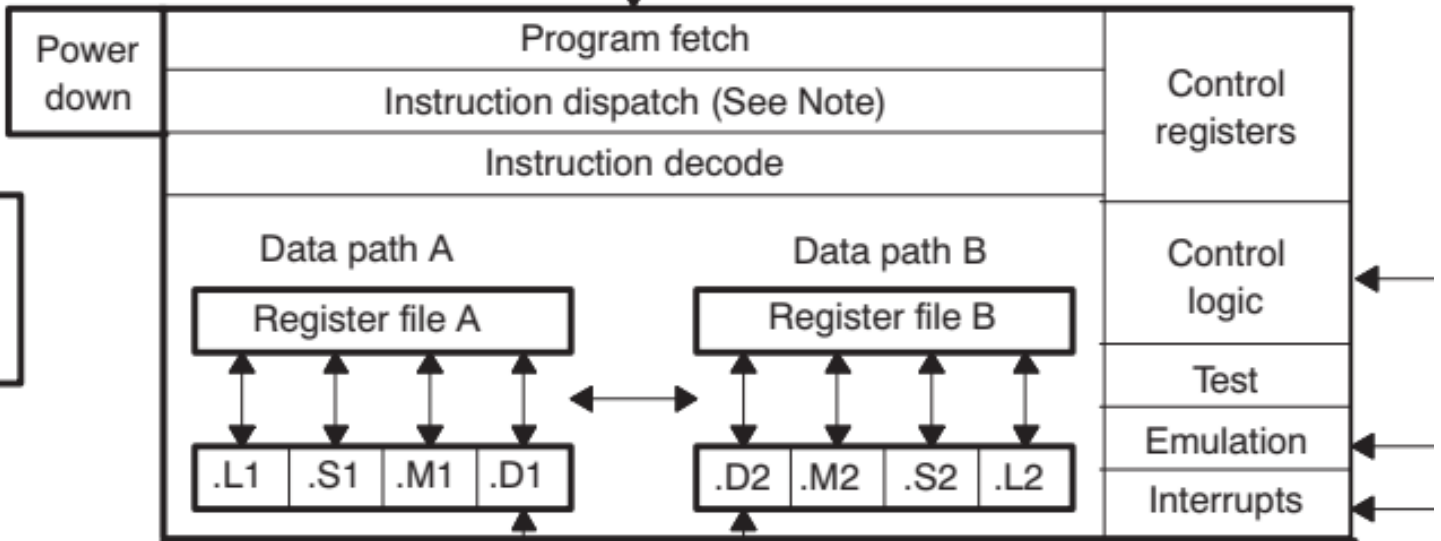


**P** = Predicate bit:  
(One bit for each 32-bit instruction)

- **Predicate bit**: Setting this bit to 1 indicates the corresponding 32-bit instruction can execute in parallel with the other instructions in the 8-word fetch packet. (**Note**: somewhat simplified description)

Program cache/program memory  
32-bit address  
256-bit data

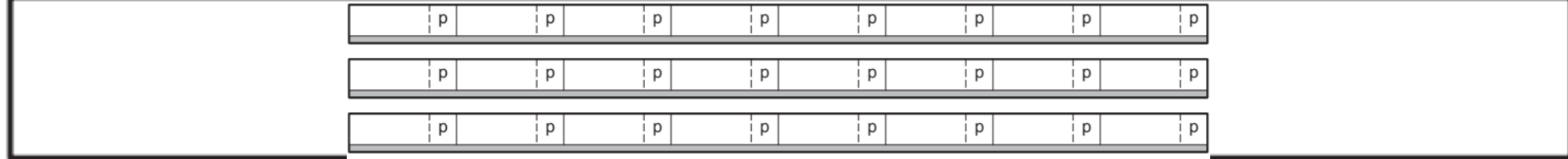
256 C6000 CPU



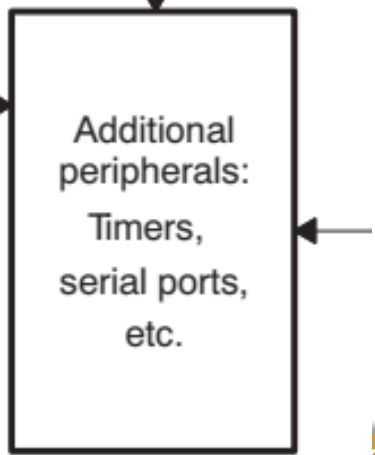
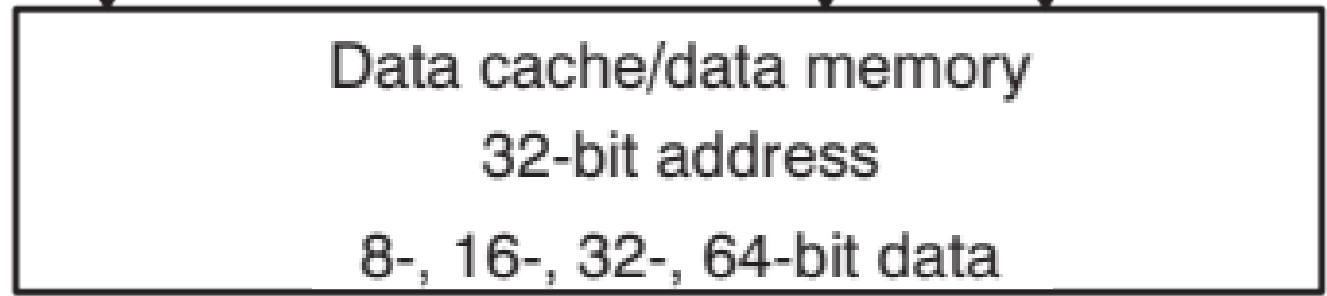
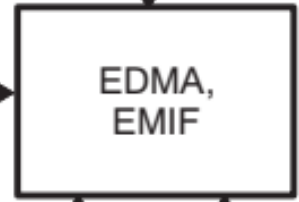
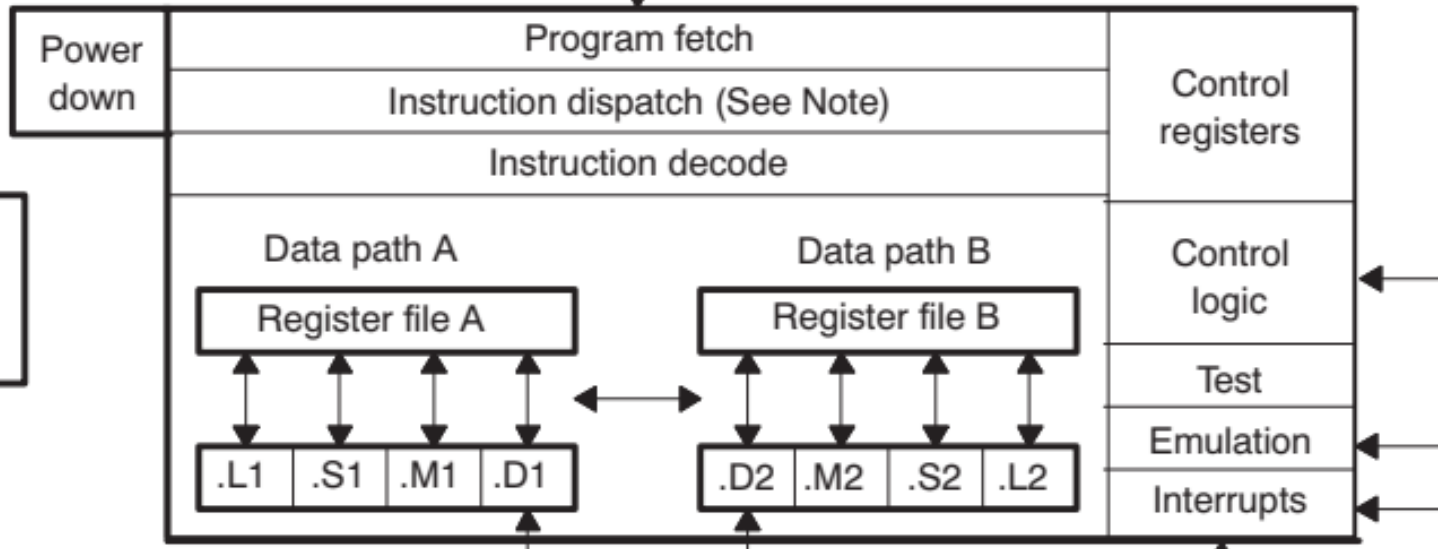
EDMA,  
EMIF

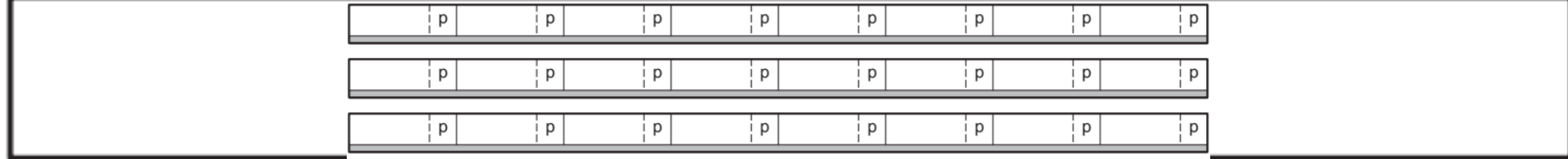
Data cache/data memory  
32-bit address  
8-, 16-, 32-, 64-bit data

Additional peripherals:  
Timers,  
serial ports,  
etc.

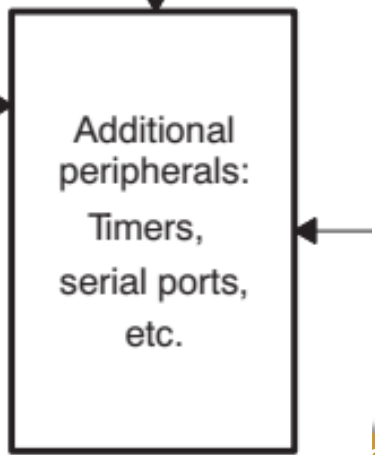
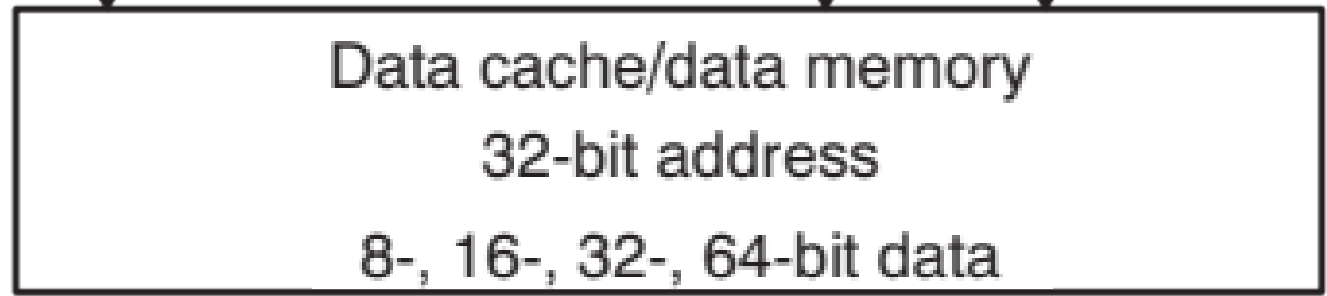
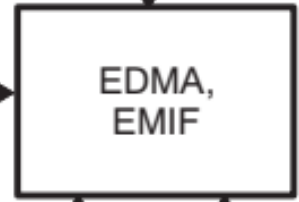
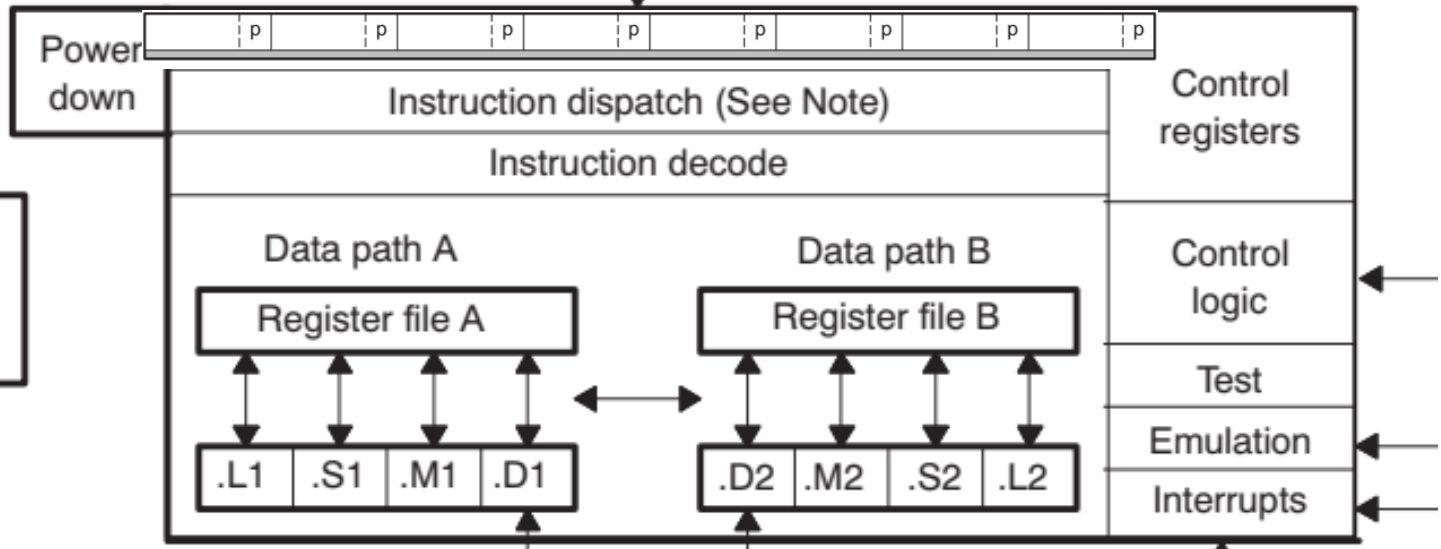


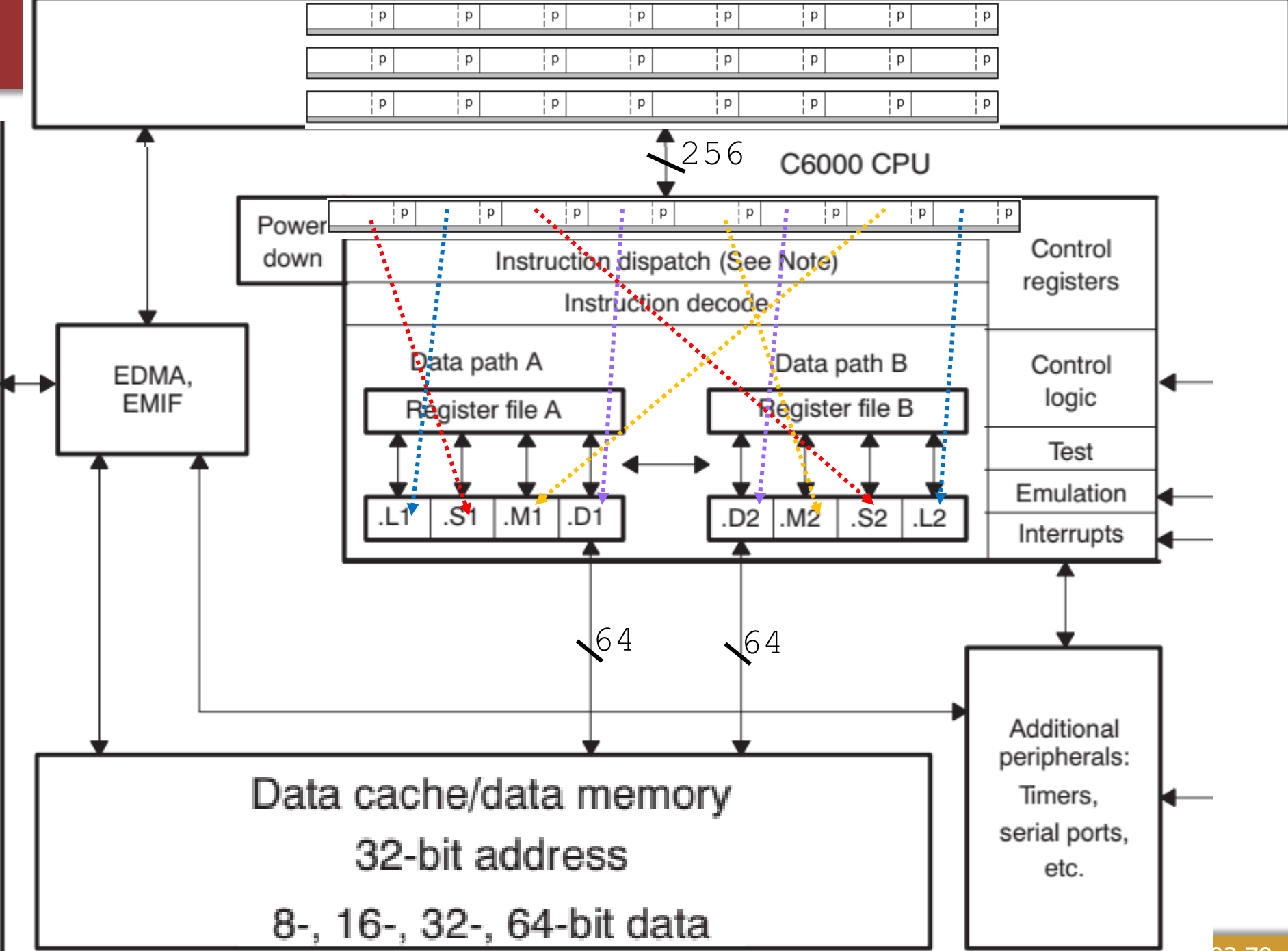
256 C6000 CPU

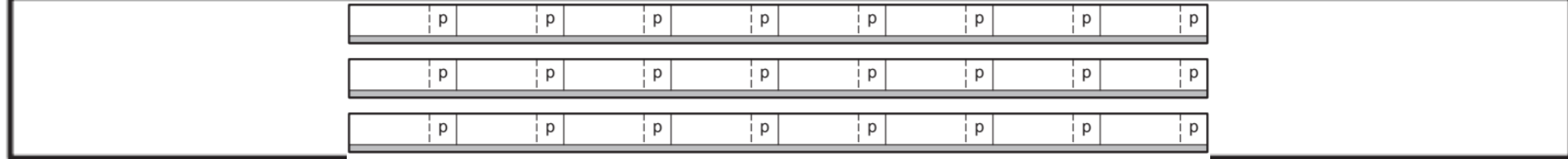




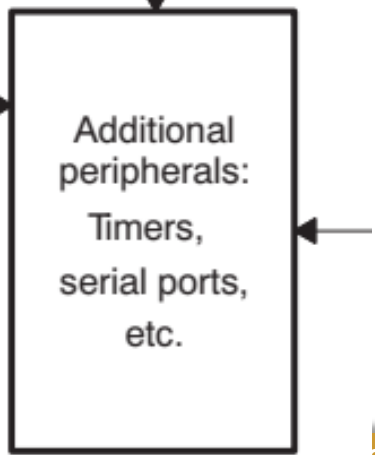
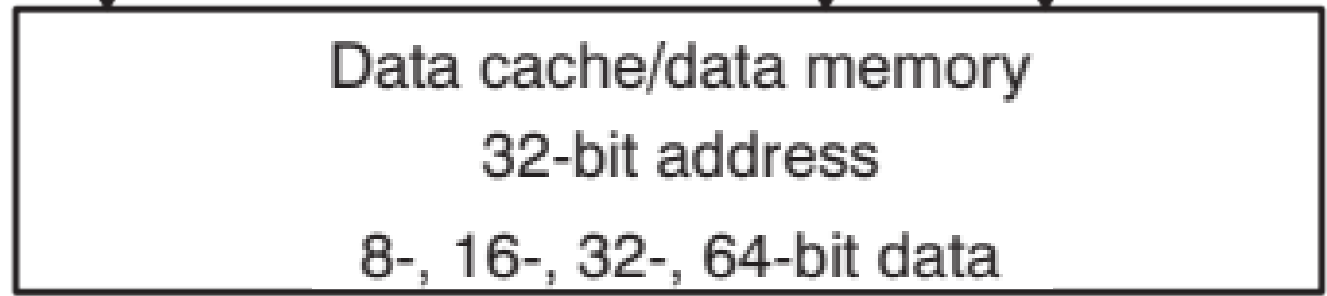
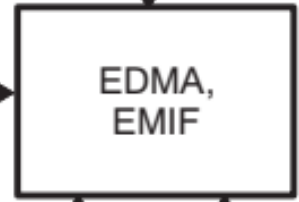
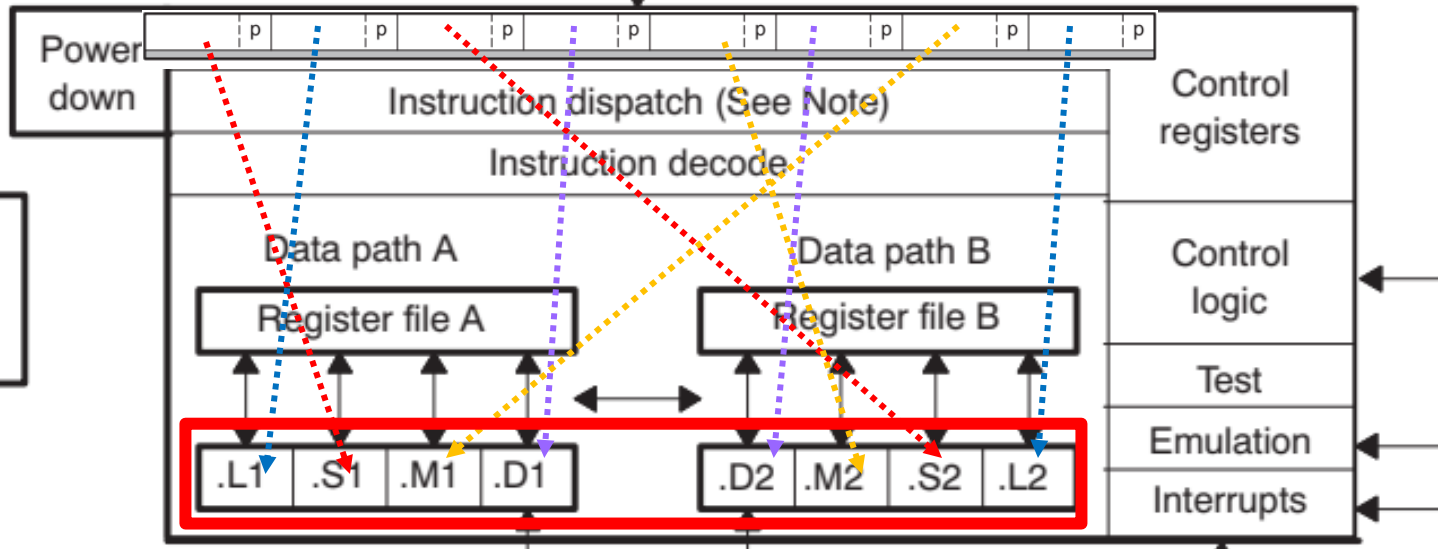
256 C6000 CPU







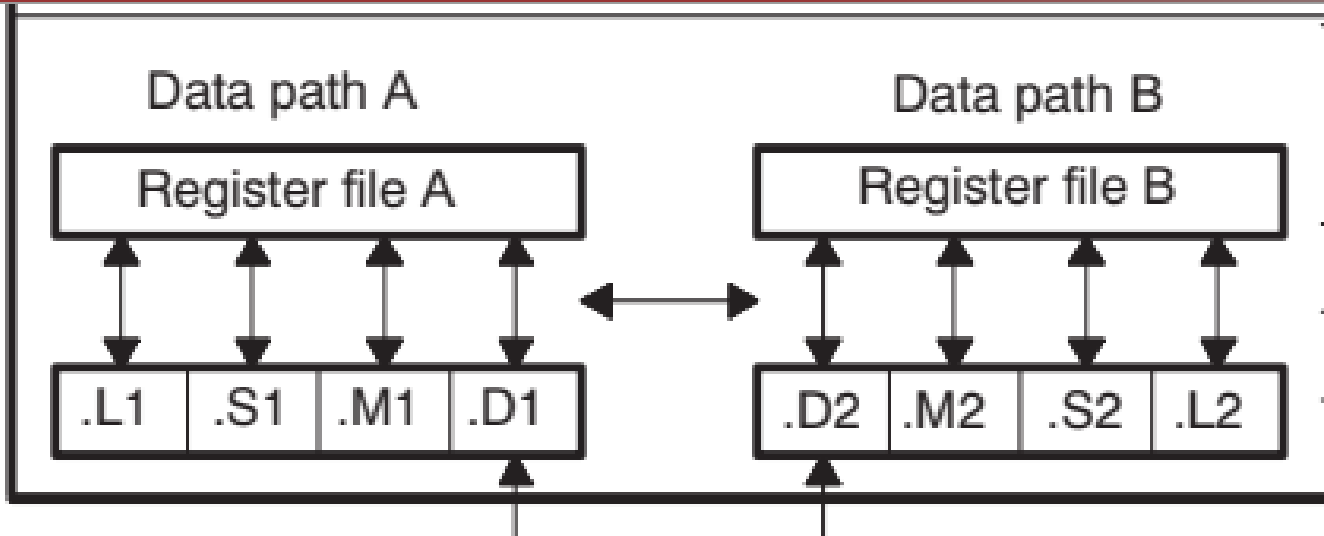
256 C6000 CPU



# C64x DSP (VLIW Computing Architecture)

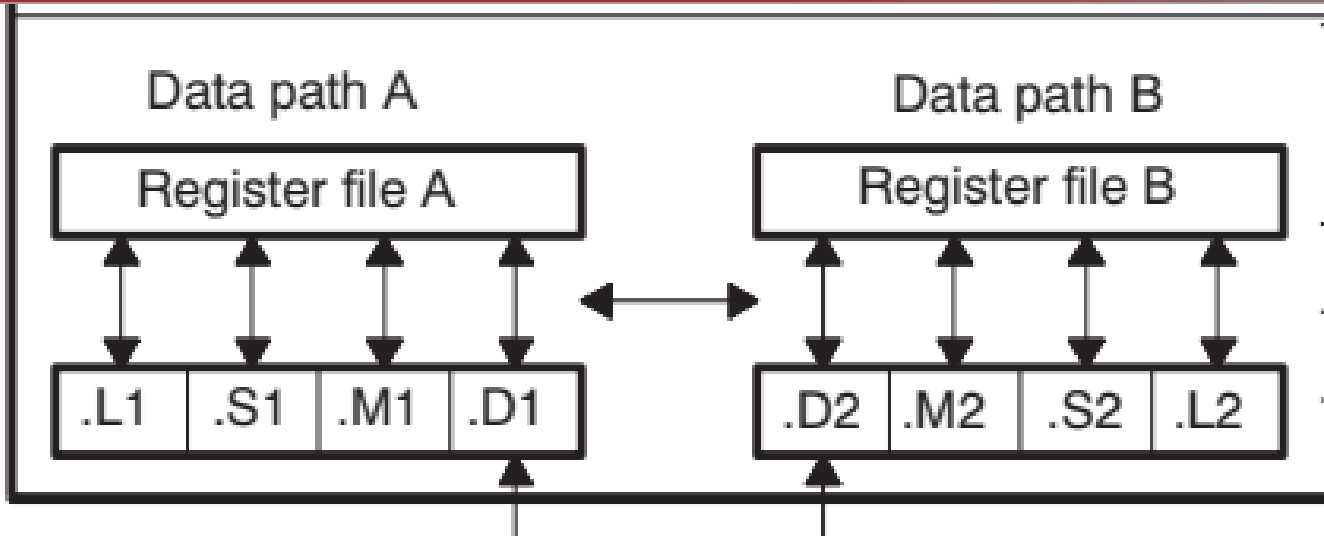
Functional Unit	Fixed-Point Operations	Functional Unit	Fixed-Point Operations
.L unit (.L1, .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits Byte shifts Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic operations Quad 8-bit arithmetic operations Dual 16-bit minimum/maximum operations Quad 8-bit minimum/maximum operations	.M unit (.M1, .M2)	32 × 32-bit multiply operations 16 × 16-bit multiply operations 16 × 32-bit multiply operations Quad 8 × 8-bit multiply operations Dual 16 × 16-bit multiply operations Dual 16 × 16-bit multiply with add/subtract operations Quad 8 × 8-bit multiply with add operation Bit expansion Bit interleaving/de-interleaving Variable shift operations Rotation Galois Field Multiply
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file Byte shifts Data packing/unpacking Dual 16-bit compare operations Quad 8-bit compare operations Dual 16-bit shift operations Dual 16-bit saturated arithmetic operations Quad 8-bit saturated arithmetic operations	.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit constant offset Loads and stores with 15-bit constant offset (.D2 only) Load and store doublewords with 5-bit constant Load and store nonaligned words and doublewords 5-bit constant generation 32-bit logical operations

# C64x DSP (VLIW Computing Architecture)



Functional Unit	Operations	Functional Unit	Operations
<b>L1, L2</b>	<ul style="list-style-type: none"> <li>• 32-bit arithmetic</li> <li>• 32-bit logic</li> <li>• Dual 16-bit arithmetic</li> <li>• Quad 8-bit arithmetic</li> </ul>	<b>M1, M2</b>	<ul style="list-style-type: none"> <li>• 32x32-bit multiply</li> <li>• Dual 16x16-bit multiply</li> <li>• Quad 8x8-bit multiply</li> </ul>
<b>S1, S2</b>	<ul style="list-style-type: none"> <li>• 32-bit arithmetic</li> <li>• 32-bit logic</li> <li>• Dual 16-bit compare</li> <li>• Quad 8-bit compare</li> </ul>	<b>D1, D2</b>	<ul style="list-style-type: none"> <li>• Double-word Store/load</li> <li>• Linear/circular address calculation</li> <li>• 32-bit logic</li> <li>• 32-bit add</li> </ul>

# C64x DSP (VLIW Computing Architecture)



Functional Unit	Operations	Functional Unit	Operations
<b>L1, L2</b>	<ul style="list-style-type: none"> <li>• 32-bit arithmetic</li> <li>• 32-bit logic</li> <li>• Dual 16-bit arithmetic</li> <li>• Quad 8-bit arithmetic</li> </ul>	<b>M1, M2</b>	<ul style="list-style-type: none"> <li>• 32x32-bit multiply</li> <li>• Dual 16x16-bit multiply</li> <li>• Quad 8x8-bit multiply</li> </ul>
<b>S1, S2</b>	<ul style="list-style-type: none"> <li>• 32-bit arithmetic</li> <li>• 32-bit logic</li> <li>• Dual 16-bit compare</li> <li>• Quad 8-bit compare</li> </ul>	<b>D1, D2</b>	<ul style="list-style-type: none"> <li>• Double-word Store/load</li> <li>• Linear/circular address calculation</li> <li>• 32-bit logic</li> <li>• 32-bit add</li> </ul>

# 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