

Instruction sets



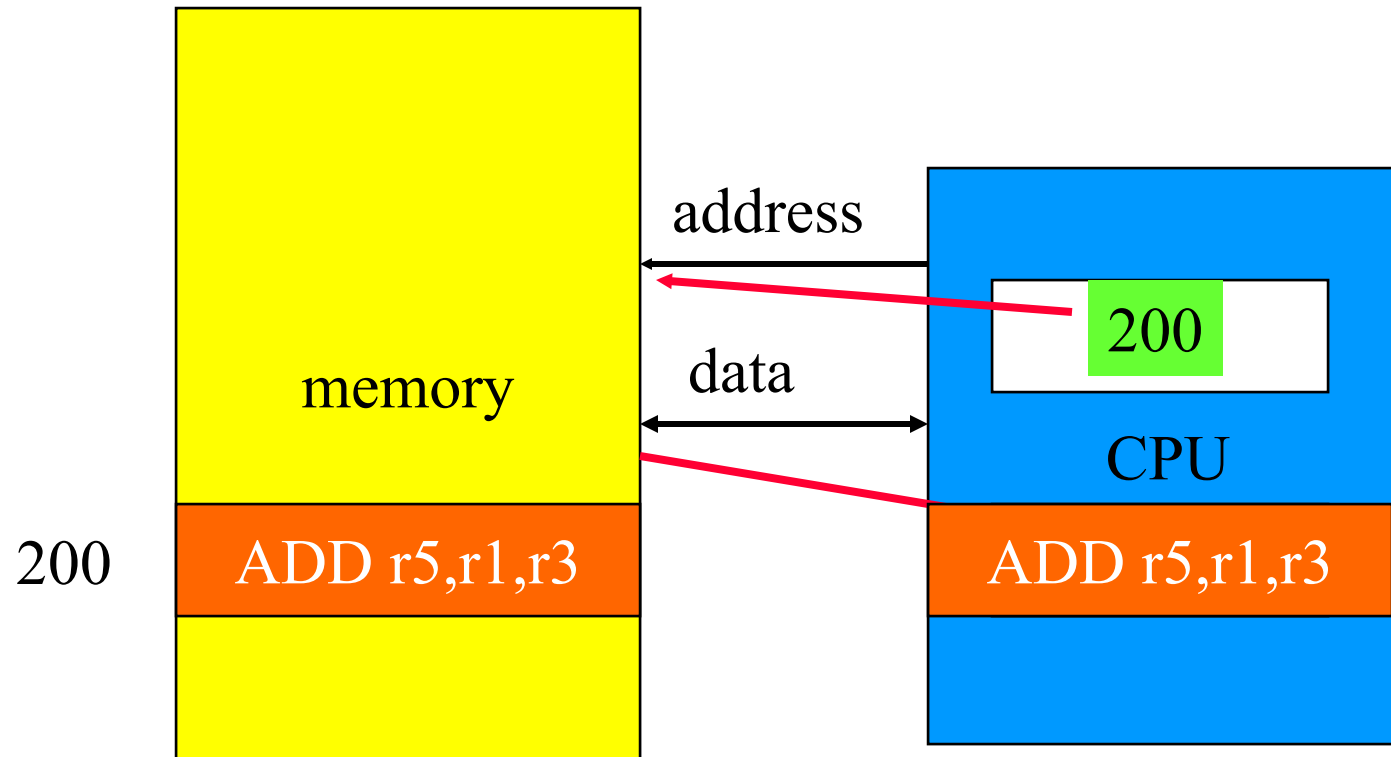
- Computer architecture taxonomy.
- Assembly language.

von Neumann architecture

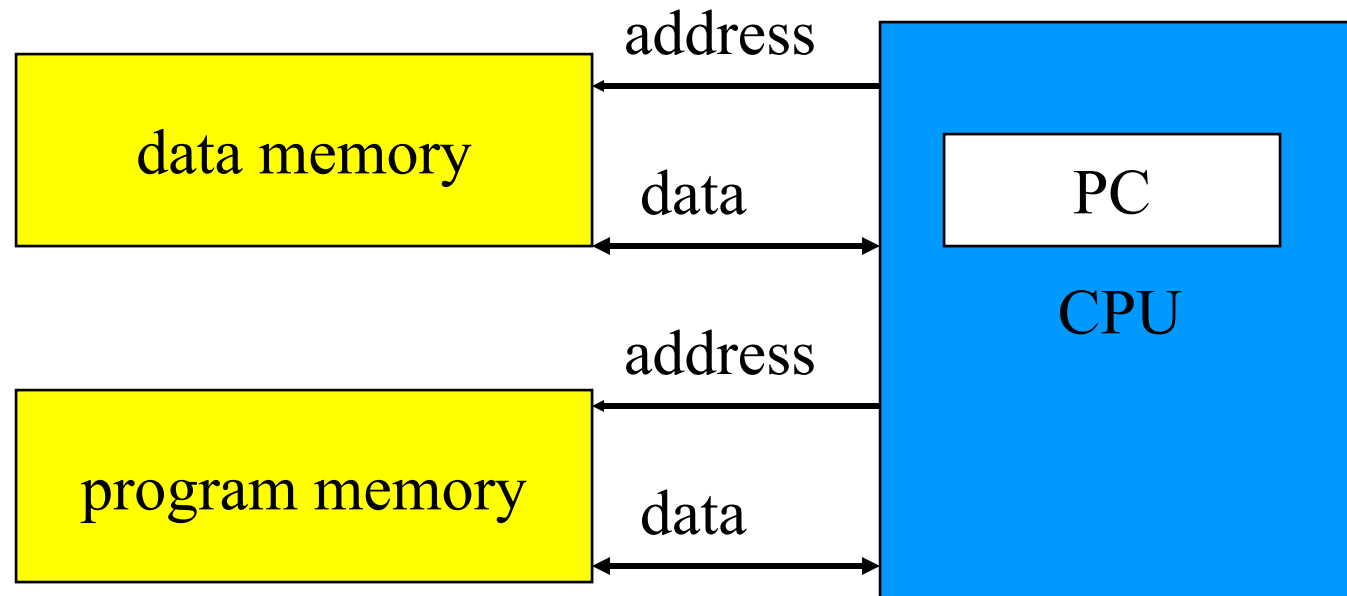


- Memory holds data, instructions.
- Central processing unit (CPU) fetches instructions from memory.
 - Separate CPU and memory distinguishes programmable computer.
- CPU registers help out: program counter (PC), instruction register (IR), general-purpose registers, etc.

CPU + memory



Harvard architecture

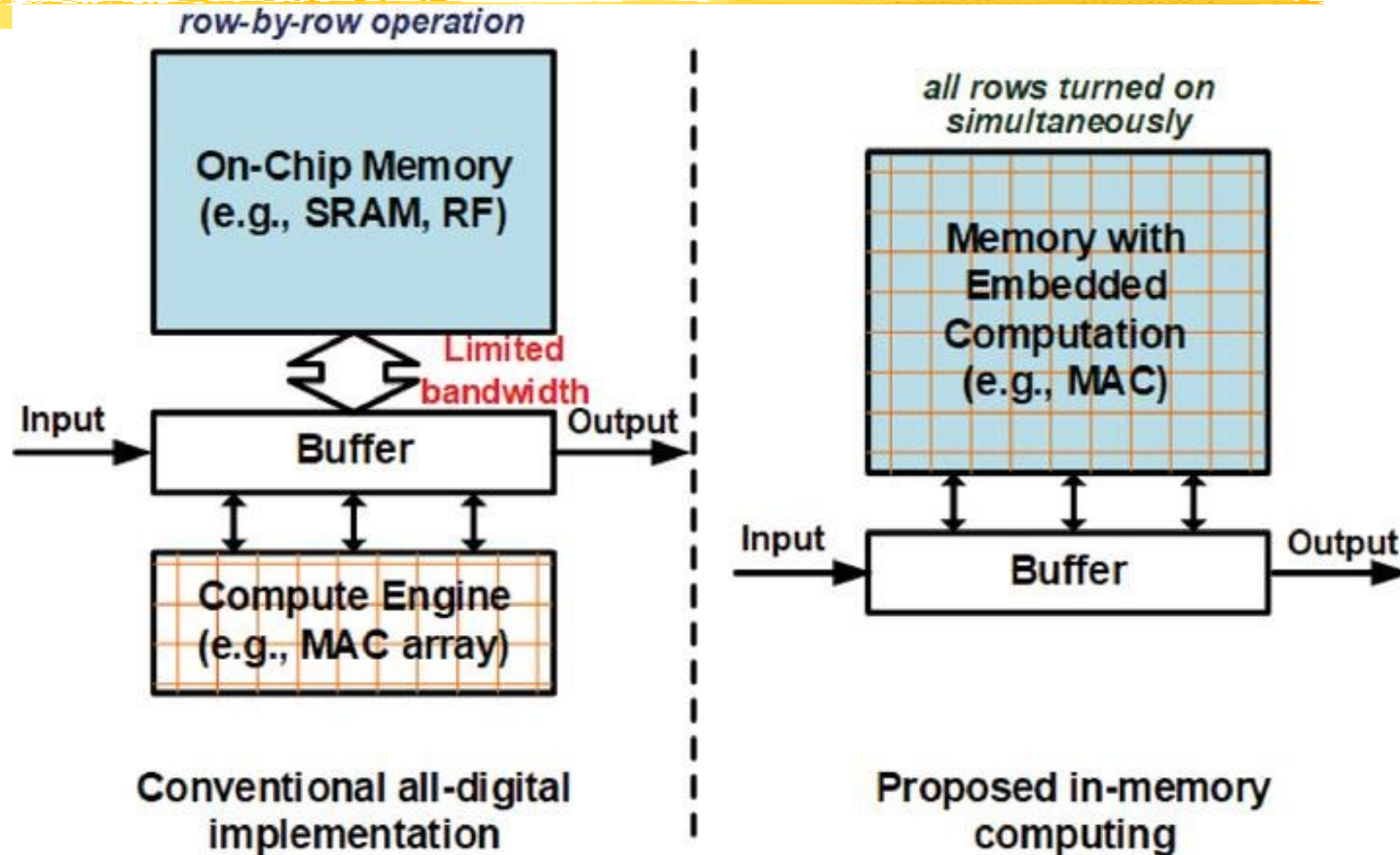


von Neumann vs. Harvard



- Harvard can't use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
 - greater memory bandwidth;
 - more predictable bandwidth.

Computing in Memory



RISC vs. CISC



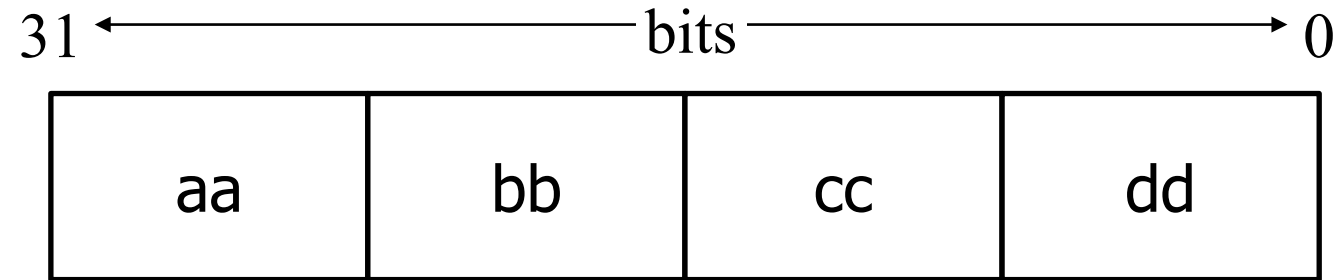
- Complex instruction set computer (CISC):
 - many addressing modes;
 - many operations.
- Reduced instruction set computer (RISC):
 - load/store;
 - pipelinable instructions.

Instruction set characteristics

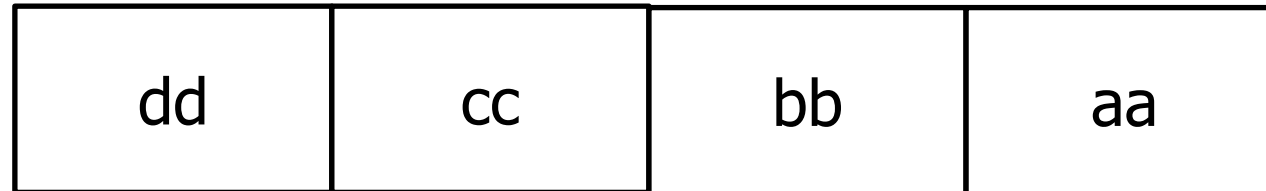


- Fixed vs. variable length.
- Addressing modes.
- Number of operands.
- Types of operands.

Big-endian vs. little-endian



big-endian



little-endian

value = 0xaabbccdd

Programming model



- **Programming model:** registers visible to the programmer.
- Some registers are not visible (IR).

Multiple implementations



- Successful architectures have several implementations:
 - varying clock speeds;
 - different bus widths;
 - different cache sizes;
 - etc.

Assembly language



- One-to-one with instructions (more or less).
- Basic features:
 - One instruction per line.
 - Labels provide names for addresses (usually in first column).
 - Instructions often start in later columns.
 - Columns run to end of line.

ARM assembly language example

```
label1  ADR  r4,c
        LDR  r0,[r4] ; a comment
        ADR  r4,d
        LDR  r1,[r4]
        SUB  r0,r0,r1 ; comment
```

Pseudo-ops



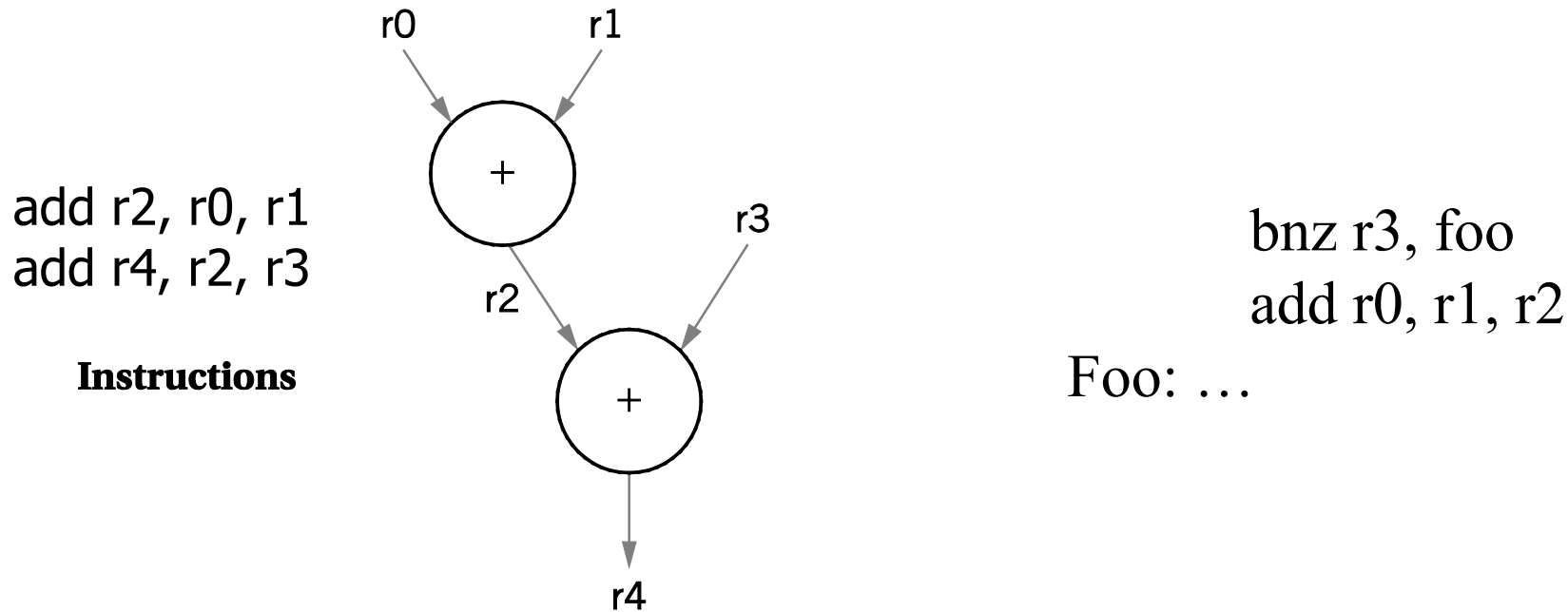
- Some assembler directives don't correspond directly to instructions:
 - Define current address.
 - Reserve storage.
 - Constants.

VLIW



- VLIW: very long instruction word.
- Performs several instructions simultaneously.
 - Architecture usually restricts the combination of instructions that can be performed at once.
- Superscalar vs. VLIW:
 - Superscalar runs standard code, determines parallel operations at run time.
 - VLIW determines parallelism at compile time.
- Packet: a set of instructions to be executed in parallel.

Data and control dependencies



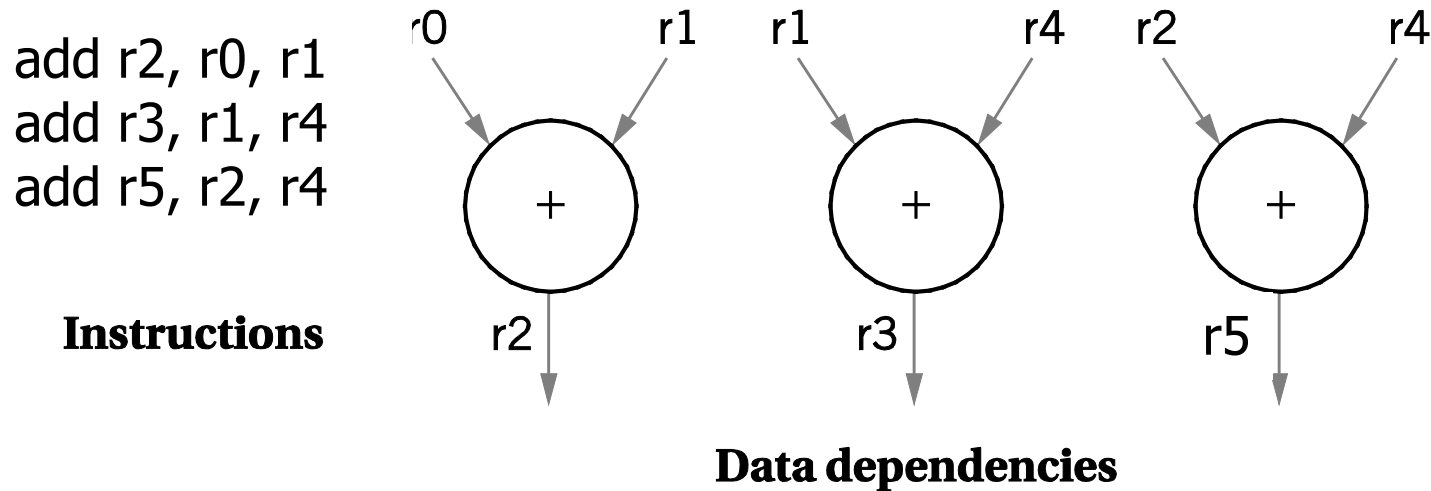
Instructions

Data dependencies

□ $r0 + r1$ must be performed before $r2 + r3$.

□ Add will be executed only if bnz fails.

Data dependencies and parallelism



- All three additions can be performed at the same time.

VLIW and embedded computing



- VLIW is more energy-efficient than superscalar.
- VLIW is widely used in signal processing.
 - Multimedia processing.
 - Processing multiple channels of signals.

ARM instruction set



- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control.

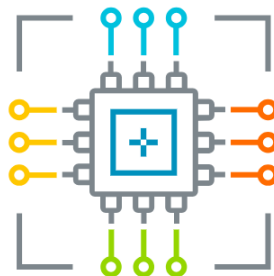
ARM versions



- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

Arm is Everywhere Technology Matters

250+ Billion Chips in Everything from Sensors to Smartphones to Servers



Processor IP >

Arm is the leading technology provider of processor IP, offering the widest range of processors to address the performance, power, and cost requirements of every device. Arm CPUs and NPUs include Cortex-A, Cortex-M, Cortex-R, Neoverse, Ethos and SecurCore.



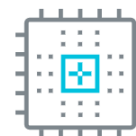
Security IP >

TrustZone, SecurCore, Cortex-M35P



System IP >

CoreLink, CoreSight, Coherent Mesh Network, AMBA and more



Graphics and Multimedia >



Software and Development

Tools >

Computers as Components

4e © 2017 Marilyn Wolf

Computers as Components 3e

© 2012 Marilyn Wolf



Products

Markets

Partners

Developers

Support & Training

Company



COMPUTE TECHNOLOGY

Cortex-X

Cortex-A

Cortex-R

Cortex-M

Ethos

Immortalis & Mali

Neoverse

Neoverse CSS

System IP

Physical IP

Security IP

Subsystems IP

ENABLING TECHNOLOGY

Architecture

Development Tools & Software

Project Cassini

Project Centauri

SystemReady



Licensing Arm Technology

Learn about our different licensing models, determine which is the best fit for your company, and connect with an Arm representative.

[View Products »](#)

Products

Arm is the world's leading semiconductor IP company. We develop and license IP that is at the heart of billions of devices worldwide, and we provide development tools that accelerate product development, from sensors to smartphones to servers.

Processor IP



Processors >

Arm is the industry's leading supplier of microprocessor technology, offering the widest range of microprocessor cores to address the performance, power and cost requirements for almost all application markets.



Multimedia >

Arm graphics and camera technology drives the ultimate visual experience across a wide range of devices, including mass-market to high-performance smartphones, Android OS-based tablets, and digital televisions (DTV).



Physical IP >

Arm provides world-class foundation physical IP and processor implementation solutions to address the performance, power, and cost requirements for all application markets.



System IP >

Arm System IP enables designers to build Arm AMBA systems that are high performance, power efficient, and reliable. Configurable for many different applications, Arm System IP is the right choice for all systems, whether high-efficiency IoT endpoints or high-performance server SoCs.



Security >

Products, programs, and technology are available from Arm for securing devices from a broad spectrum of potential vulnerabilities. Arm security IP lets our partners deploy the security level that best matches application needs.



Subsystems >

SoC designers manage the creation of increasingly complex IoT and embedded devices. Arm's range of Corstone packages provide designers with a solid foundation for building secure SoCs, integrating pre-verified subsystems and system IP in one kit.

Arm CPU Architecture: A Foundation for Computing Everywhere

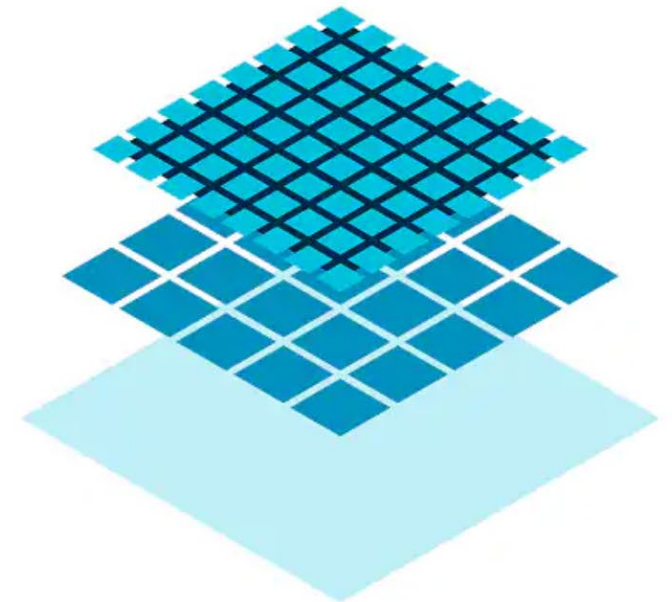
The Arm architecture is a family of reduced instruction set computing (RISC) architectures for computer processors. It is the most pervasive processor architecture in the world, with more than 250 billion Arm-based chips shipped by our partners over the past three decades in products ranging from sensors, wearables and smartphones to supercomputers. Benefits of the Arm CPU architecture include:

- Integrated security
- High performance and energy efficiency
- Large ecosystem for global support
- Pervasive across markets and locations

Implementation

Microarchitecture

Architecture





CPU architecture profiles

A-profile (Application)

Offers highest performance of all architecture profiles

[Learn More](#) >

R-profile (Real-Time)

Optimized for systems with real-time requirements

[Learn More](#) >

M-profile (Microcontroller)

Designed for small, low power, highly energy-efficient devices

[Learn More](#) >

Implementations

Arm Processor IP range:
[Cortex-A](#), [Neoverse](#), and [Cortex-X](#)

Arm Processor IP range:
[Cortex-R](#)

Arm Processor IP range:
[Cortex-M](#)

Features

- Offers highest performance of all architecture profiles
- Highly energy efficient
- Optimized to run rich operating systems

- Optimized for systems with real-time requirements

- Designed for small, low power, highly energy-efficient devices

Latest Versions

Armv9-A and Armv8-A

Armv8-R

Armv8-M

Use Cases

Complex compute application areas, such as PCs, laptops, smart TVs, servers, networking equipment, smartphones and automotive head units, cloud storage, and supercomputers.

Real-time response requirements in safety critical applications or applications needing a deterministic response, such as medical equipment, vehicle steering, braking and signalling, networking and storage equipment, and embedded control systems.

Energy efficiency, power consumption, and size priorities. Security processors, IoT and embedded devices, such as wearables, small sensors, communication modules and smart home products.

ARM family



- Cortex-A: built for advanced operating systems and exhibits the highest possible performance;
- Cortex-R: caters perfectly to the needs of real-time applications and provides its users with the fastest response times;
- Cortex-M: mainly built for microcontrollers;
- Cortex-X: The Arm Cortex-X Custom Program enables customization and differentiation beyond the traditional roadmap of Arm Cortex products, offering a way to deliver the ultimate performance required for the use cases of tomorrow.

Architecture ↕	Core bit-width ↕	Cores		Profile ↕
		Arm Ltd. ↕	Third-party ↕	
ARMv1	32	ARM1		Classic
ARMv2	32	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core ^[63]	Classic
ARMv3	32	ARM6, ARM7		Classic
ARMv4	32	ARM8	StrongARM, FA526, ZAP Open Source Processor Core	Classic
ARMv4T	32	ARM7TDMI, ARM9TDMI, SecurCore SC100		Classic
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	Classic
ARMv6	32	ARM11		Classic
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300	Apple M7 motion coprocessor	Microcontroller
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller
ARMv8-M	32	ARM Cortex-M23, ^[65] ARM Cortex-M33 ^[66]		Microcontroller
ARMv8.1-M	32	ARM Cortex-M55, ARM Cortex-M85		Microcontroller
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time
ARMv8-R	32	ARM Cortex-R52		Real-time
	64	ARM Cortex-R82 ↗		Real-time
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Scorpion/Krait, PJ4/Sheeva, Apple Swift (A6, A6X)	Application
ARMv8-A	32	ARM Cortex-A32 ^[72]		Application
	64/32	ARM Cortex-A35, ^[73] ARM Cortex-A53, ARM Cortex-A57, ^[74] ARM Cortex-A72, ^[75] ARM Cortex-A73 ^[76]	X-Gene, Nvidia Denver 1/2, Cavium ThunderX, AMD K12, Apple Cyclone (A7)/Typhoon (A8, A8X)/Twister (A9, A9X)/Hurricane+Zephyr (A10, A10X), Qualcomm Kryo, Samsung M1/M2 ("Mongoose") /M3 ("Meerkat")	Application

SystemReady



CPU



GIC



AMBA



SMMU



Debug
Trace



Security



Software Standards



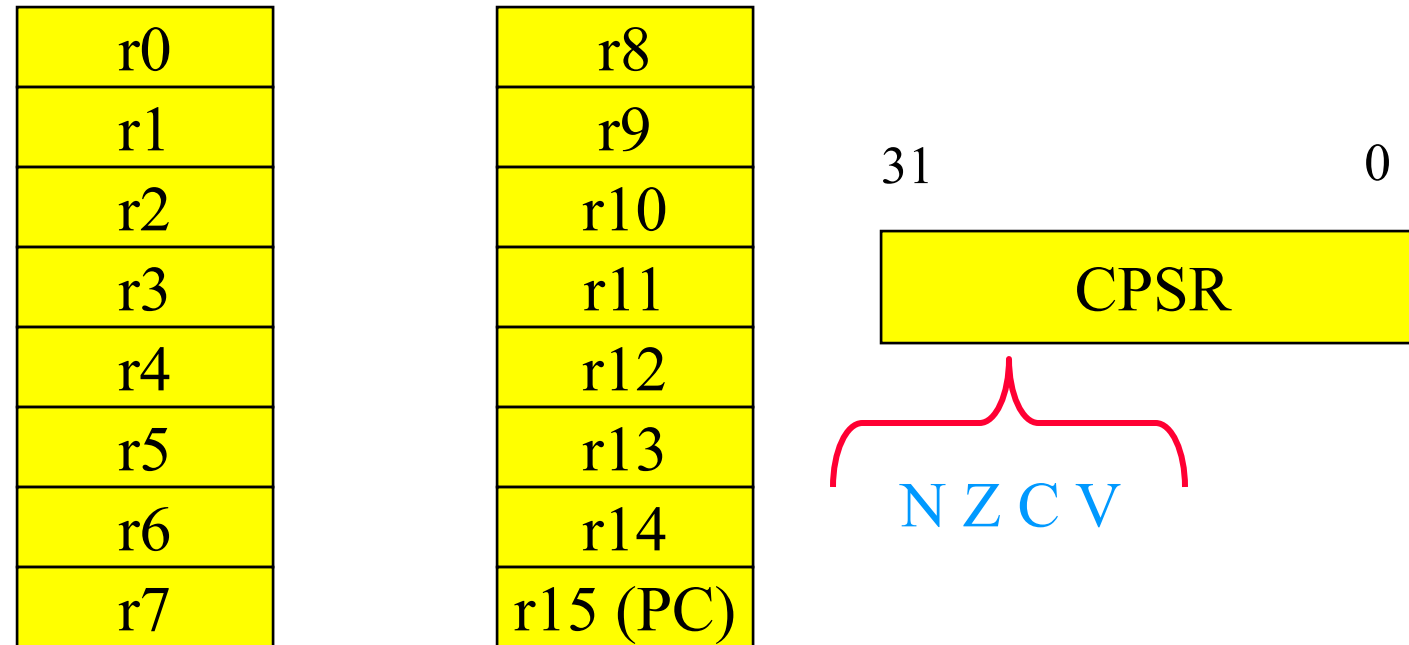
ARM assembly language



- Fairly standard assembly language:

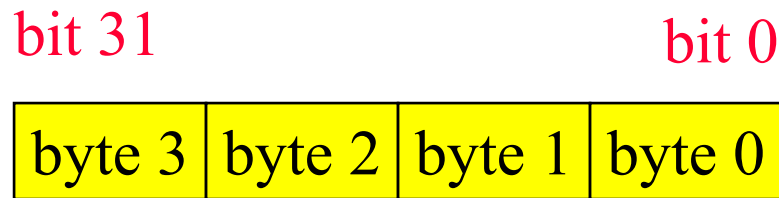
```
        LDR r0,[r8] ; a comment
label1  ADD r4,r0,r1
```

ARM programming model

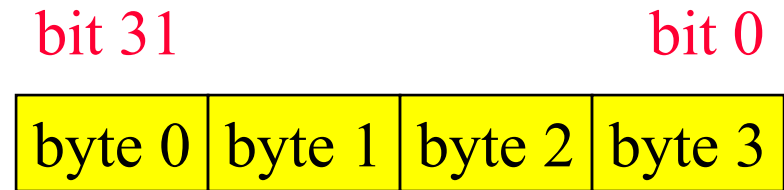


Endianness

- Relationship between bit and byte/word ordering defines endianness:



little-endian



big-endian

ARM data types



- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long. ($2^{32}=4294967296$)
- Address refers to byte.
 - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or big-endian mode.

ARM status bits

- Every arithmetic, logical, or shifting operation sets CPSR bits:
 - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
 - $-1 + 1 = 0$: NZCV = 0110. -1:0xffffffff 1:0x00000001
 - $(2^{31}-1)+1 = -2^{31}$: NZCV = 1001. 0x7fffffff + 0x1 = 0x80000000

ARM data instructions



- Basic format:

ADD r0,r1,r2

- Computes $r1+r2$, stores in r0.

- Immediate operand:

ADD r0,r1,#2

- Computes $r1+2$, stores in r0.

ARM data instructions

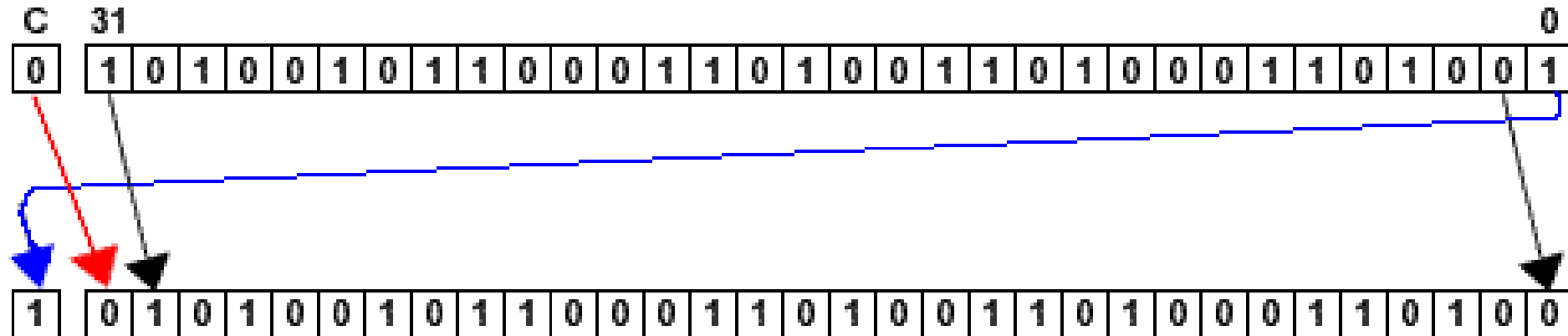
- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
 - SUB $r0, r1, r2 \rightarrow r0 = r1 - r2$
 - RSB $r0, r1, r2 \rightarrow r0 = r2 - r1$
- MUL, MLA : multiply (and accumulate)
 - MLA $r0, r1, r2, r3 \rightarrow r0 = r1 \times r2 + r3$
- AND, ORR, EOR (XOR)
 - $1001 \text{ xor } 1100 = 0101$
 - $1001 \text{ orr } 1100 = 1101$
 - $1001 \text{ and } 1100 = 1000$
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties



- Logical shift:
 - fills with zeroes.
- Arithmetic shift:
 - fills with ones.
- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.

RRX example



RRX

ARM comparison instructions



- CMP : compare
- CMN : negated compare
- TST : bit-wise test
- TEQ : bit-wise negated test
- These instructions set only the NZCV bits of CPSR.

ARM move instructions



□ MOV, MVN : move (negated)

MOV r0, r1 ; sets r0 to r1

ARM load/store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
 - register indirect : LDR r0, [r1]
 - with second register : LDR r0, [r1, -r2]
 - with constant : LDR r0, [r1, #4]

ARM ADR pseudo-op



- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

```
ADR r1, F00
```

Example: C assignments

□ C:

```
x = (a + b) - c;
```

□ Assembler:

```
ADR r4,a           ; get address for a
LDR r0,[r4]         ; get value of a
ADR r4,b           ; get address for b, reusing r4
LDR r1,[r4]         ; get value of b
ADD r3,r0,r1        ; compute a+b
ADR r4,c           ; get address for c
LDR r2[r4]          ; get value of c
```

C assignment, cont'd.



```
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
STR r3[r4] ; store value of x
```

Example: C assignment

□ C:

```
y = a*(b+c);
```

□ Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.



```
MUL r2,r2,r0 ; compute final value for y  
ADR r4,y ; get address for y  
STR r2,[r4] ; store y
```

Example: C assignment

□ C:

```
z = (a << 2) | (b & 15);
```

□ Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
```

C assignment, cont'd.



```
ADR r4,z ; get address for z  
STR r1,[r4] ; store value for z
```

Additional addressing modes

- Base-plus-offset addressing:

LDR r0, [r1, #16]

- Loads from location r1+16

- Auto-indexing increments base register: for(i=0; i<x; i++/++i)

LDR r0, [r1, #16]!

- Post-indexing fetches, then does offset:

LDR r0, [r1], #16

- Loads r0 from r1, then adds 16 to r1.

ARM flow of control



- All operations can be performed conditionally, testing CPSR:
 - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Branch operation:
 - B #100
 - Can be performed conditionally.

Example: if statement

□ C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

□ Assembler:

```
; compute and test condition
```

```
ADR r4,a ; get address for a
```

```
LDR r0,[r4] ; get value of a
```

```
ADR r4,b ; get address for b
```

```
LDR r1,[r4] ; get value for b
```

```
CMP r0,r1 ; compare a < b
```

```
BGE fblock ; if a >= b, branch to false block
```

If statement, cont'd.



```
; true block
MOV r0,#5 ; generate value for x
ADR r4,x ; get address for x
STR r0,[r4] ; store x
ADR r4,c ; get address for c
LDR r0,[r4] ; get value of c
ADR r4,d ; get address for d
LDR r1,[r4] ; get value of d
ADD r0,r0,r1 ; compute y
ADR r4,y ; get address for y
STR r0,[r4] ; store y
B after ; branch around false block
```

If statement, cont'd.



```
; false block
fblock ADR r4,c ; get address for c
      LDR r0,[r4] ; get value of c
      ADR r4,d ; get address for d
      LDR r1,[r4] ; get value for d
      SUB r0,r0,r1 ; compute a-b
      ADR r4,x ; get address for x
      STR r0,[r4] ; store value of x
after ...
```

Example: switch statement

□ C:

```
switch (test) { case 0: ... break; case 1: ... }
```

□ Assembler:

```
ADR r2,test ; get address for test
```

```
LDR r0,[r2] ; load value for test
```


```
ADR r1,switchtab ; load address for switch table
```

```
LDR r1,[r1,r0,LSL #2] ; index switch table    r1 = r1 + r0(test)*4
```

```
switchtab DCD case0
```

```
DCD case1
```

```
...
```

- 
- The DCD directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory. & is a synonym for DCD.

- &a, &b

Syntax

The syntax of `DCD` is:

```
{label} DCD expression{,expression}
```

where:

`expression`

is either:

- A numeric expression. See [Numeric expressions](#).
- A program-relative expression.

Example: FIR filter

□ C:

```
for (i=0, f=0; i<N; i++)  
    f = f + c[i]*x[i];
```

□ Assembler

; loop initiation code

MOV r0,#0 ; use r0 for I

MOV r8,#0 ; use separate index for arrays

ADR r2,N ; get address for N

LDR r1,[r2] ; get value of N

MOV r2,#0 ; use r2 for f

FIR filter, cont'.d



```
ADR r3,c ; load r3 with base of c
ADR r5,x ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8] ; get c[i]
    LDR r6,[r5,r8] ; get x[i]
    MUL r4,r4,r6 ; compute c[i]*x[i]
    ADD r2,r2,r4 ; add into running sum
    ADD r8,r8,#4 ; add one word offset to array index
    ADD r0,r0,#1 ; add 1 to i
    CMP r0,r1 ; exit?
    BLT loop ; if i < N, continue
```


ARM subroutine linkage

- Branch and link instruction:

BL foo

- Copies current PC to r14.

- To return from subroutine:

MOV r15,r14

$x = a + b;$

foo(x);

$y = c - d$

Nested subroutine calls

□ Nesting/recursion requires coding convention:

```
f1    LDR r0,[r13] ; load arg into r0 from stack
      ; call f2()
      STR r13!, [r14] ; store f1's return adrs
      STR r13!, [r0] ; store arg to f2 on stack
      BL f2 ; branch and link to f2
      ; return from f1()
      SUB r13, #4 ; pop f2's arg off stack
      LDR r13!, r15 ; restore register and return
```

Summary



- Load/store architecture
- Most instructions are RISC, operate in single cycle.
 - Some multi-register operations take longer.
- All instructions can be executed conditionally.

PICmicro PIC16F



- PIC16F processor and memory organization.
- PIC16F data operations.
- PIC16F flow of control.



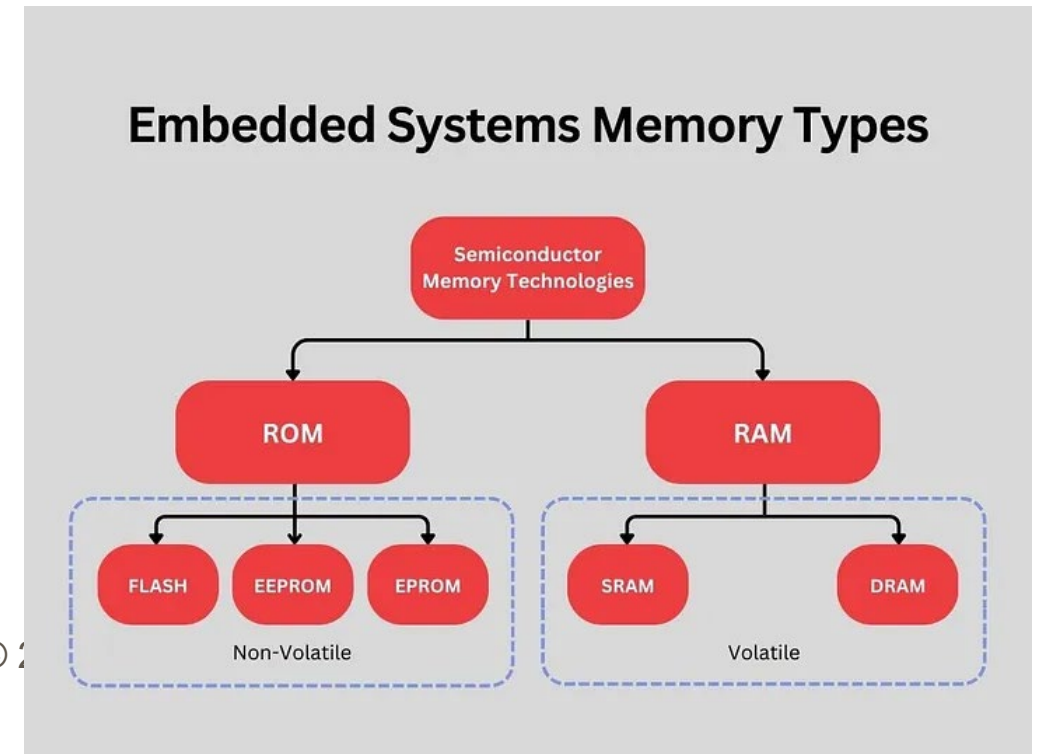
Processor and memory organization



- Exact features vary from model to model.
- Harvard architecture.
- Instruction flash memory up to 8192 words.
 - Instructions are 14 bits long.
- Data memory includes up to 368 words SRAM and 256 bytes EEPROM.
 - Data is byte-addressable.

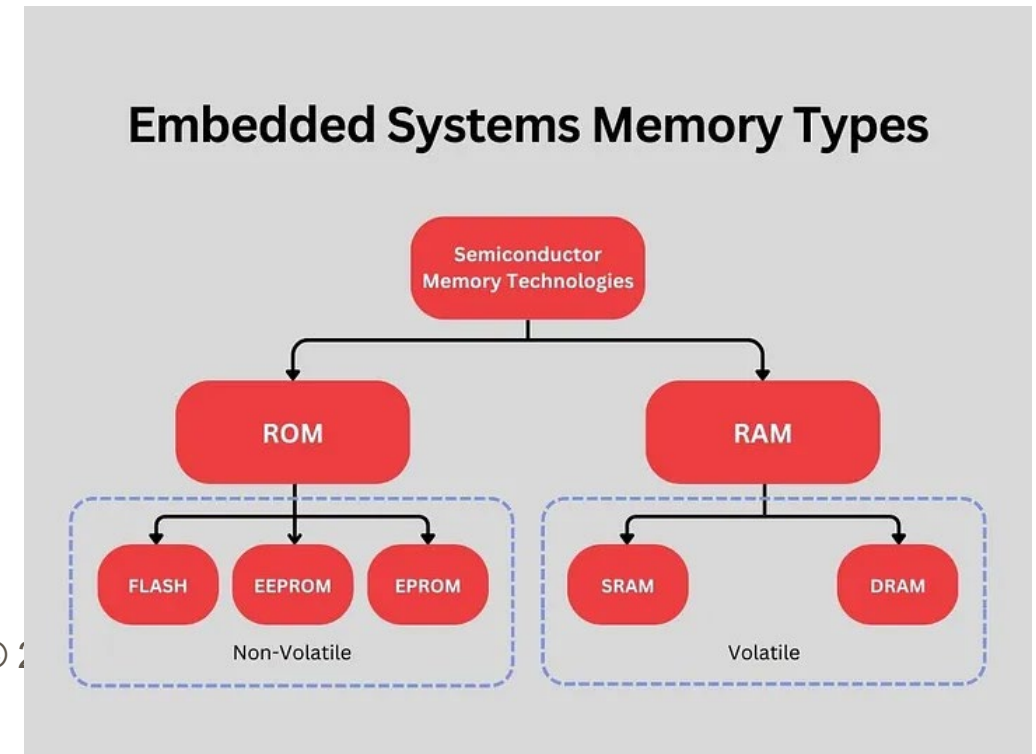
Memory

- ❑ Flash memory is a non-volatile, electrically reprogrammable storage solution. It offers high-density, low-cost storage with fast read times. Flash memory is widely used in embedded systems due to its numerous advantages, such as its ability to maintain data without power and quick access to stored data.
- ❑ Pros:
 - ❑ High-density storage
 - ❑ Low cost
 - ❑ Fast read times
 - ❑ Non-volatile, retaining data without power
 - ❑ Electrically reprogrammable
- ❑ Cons:
 - ❑ Erasing data is limited to one sector at a time
 - ❑ Slower write times compared to RAM
 - ❑ Finite number of write/erase cycles



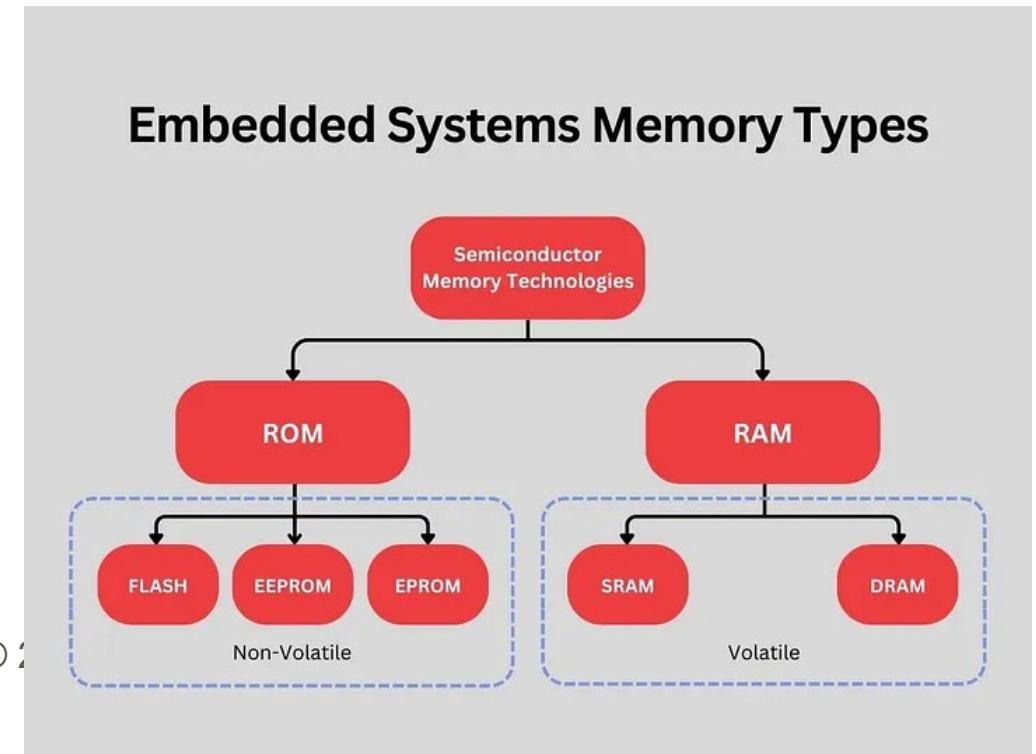
Memory

- ❑ Static RAM (SRAM) is a type of volatile memory that provides fast access times. It retains data as long as power is supplied to the system. SRAM is commonly used for high-speed caches in embedded systems.
- ❑ Pros:
 - ❑ Fast access times
 - ❑ No refresh cycles required, unlike DRAM
- ❑ Cons:
 - ❑ Higher cost-per-byte compared to DRAM
 - ❑ Consumes more power than DRAM
 - ❑ Requires more transistors per memory cell,
 - ❑ resulting in a larger chip size



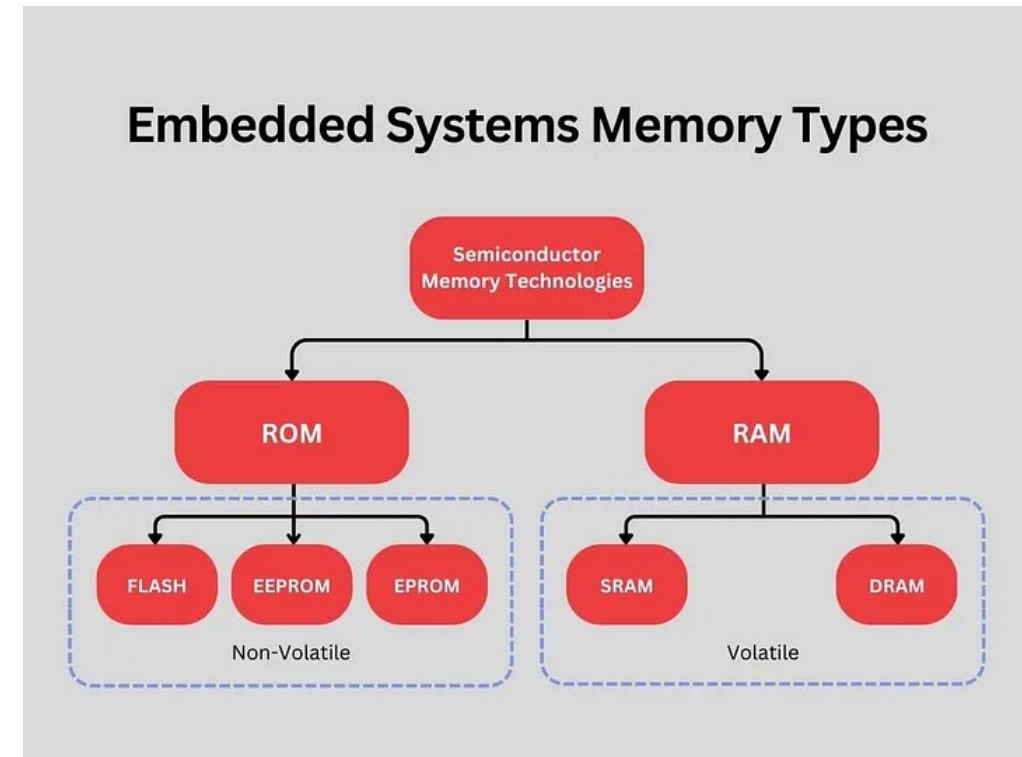
Memory

- ❑ Electrically-Erasable-Programmable Read-Only Memory (EEPROM) is a hybrid memory device that combines features of both RAM and ROM. It can be read and written like RAM, but it retains its contents without power, like ROM. EEPROMs can be erased and reprogrammed electrically, offering more flexibility than EPROMs.
- ❑ Pros:
 - ❑ Non-volatile, retaining data without power
 - ❑ Electrically erasable and reprogrammable
 - ❑ Allows byte-by-byte erasing and writing
- ❑ Cons:
 - ❑ Higher cost compared to other memory types
 - ❑ Longer write cycles than RAM
 - ❑ Limited number of write/erase cycles

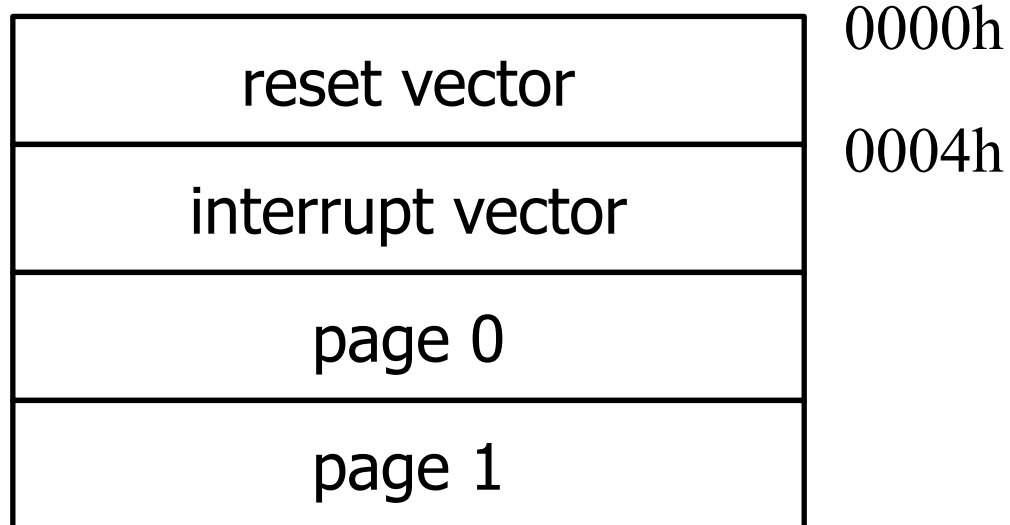
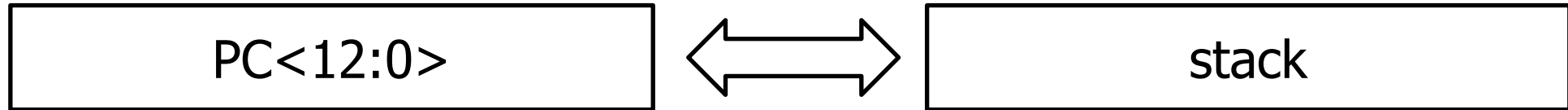


Memory

- ❑ Flash memory:
 - ❑ Widely used in embedded systems for firmware storage, data logging, and file systems due to its non-volatile nature, high-density storage, and low cost.
- ❑ SRAM:
 - ❑ Often employed as cache memory or for high-speed data paths in embedded systems, thanks to its fast access times. It is also suitable for low-power applications due to its simple internal structure.
- ❑ EEPROM:
 - ❑ Typically utilized for storing small amounts of non-volatile data, such as configuration settings, calibration data, and device IDs. Its byte-by-byte erasability and reprogrammability allow for easy updates and modifications.



Addresses and instructions



...

Program counter and status register

- PC can be loaded from 8-level stack.
 - Separate from program and data memory.
- Stack operates as a circular buffer.
- Used by CALL, RETURN/RETLW/RETFIE.
- STATUS located in bank 0.
 - ALU status, reset, bank select.

Data space



- General purpose register: data memory location.
- Special function register: I/O devices, etc.

Addressing modes



- Terminology:
 - f: general-purpose register.
 - W: accumulator.
 - b: bit address in register.
 - k: literal, constant, or label.
- Indirect addressing controlled by INDF and FSR registers.
 - Access to INDF causes indirect load through FSR.

Data instructions

ADDLW	Add literal and W
BCF	Bit clear f
ADDWF	Add W and f
BSF	Bit set f
ANDLW	AND literal with W
ANDWF	AND W with f
COMF	Complement f
CLRF	Clear f
DECf	Decrement f
CLRW	Clear W
IORLW	Inclusive OR literal with W
INCF	Increment f
IORWF	Inclusive OR W with F
MOVF	Move f
MOVWF	Move W to f
MOVLW	Move literal to W
NOP	No operation
RLF	Rotate left F through carry
RRF	Rotate right F through carry
SUBWF	Subtract W from F
SWAPF	Swap nibbles in F
XORLW	Exclusive OR literal with W
CLRWDt	Clear watchdog timer
SUBLW	Subtract W from literal

- Arithmetic.
- Boolean.
- Move.
- Rotate.
- Swap.
- Watchdog timer.

Flow of control



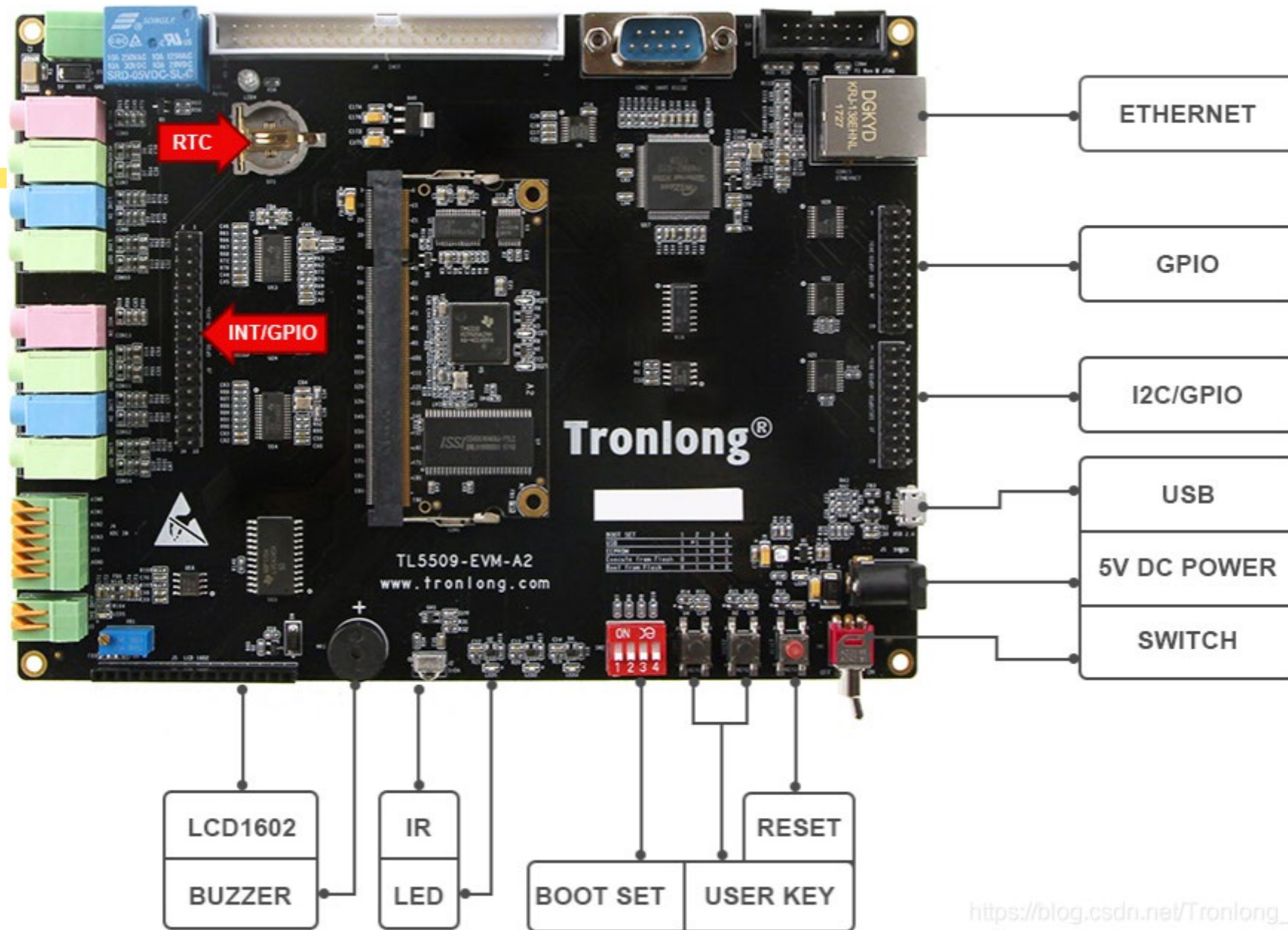
BTFSC	Bit test f, skip if clear
BTFSS	Bit test f, skip if set
CALL	Call subroutine
DECFSZ	Decrement f, skip if 0
INCFSZ	Increment f, skip if 0
GOTO	Unconditional branch
RETFIE	Return from interrupt
RETLW	Return with literal in W
RETURN	Return from subroutine
SLEEP	GO into standby mode

- Unconditional branch.
- Conditional branch.
- Subroutine call/return.
- Interrupt return.
- Standby mode.

TI C55x instruction set



- C55x programming model.
- C55x assembly language.
- C55x memory organization.
- C55x data operations.
- C55x flow of control.



https://blog.csdn.net/Tronlong_

TI C55x overview

- Accumulator architecture:

- $\text{acc} = \text{operand} + \text{acc}.$

- Very useful in loops for DSP. $y = a_1 * x_1 + a_2 * x_2 + \dots$

- C55x assembly language:

- $\text{MPY } *AR0, *CDP+, AC0$

- Label: $\text{MOV } \#1, T0$

- C55x algebraic assembly language:

- $AC1 = AR0 * \text{coef}(*CDP)$

Intrinsic functions



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

C55x data types



- Data types:
 - Word: 16 bits.
 - Longword: 32 bits.
- Instructions are byte-addressable.
- Some instructions operate on register bits.

C55x registers



- Terminology:
 - Register: any type of register.
 - Accumulator: $\text{acc} = \text{operand op ac.}$
- Most registers are memory-mapped.

C55x program counter and 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 stack pointers



- SP keeps track of user stack.
- SSP holds system stack pointer.
- SPH is extended data page pointer for both SP and SSP.

C55x auxiliary registers and circular buffer pointers



- AR0-AR7 are auxiliary instructions.
- 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 block repeat registers



- BRC9 counts block repeat instructions.
- RSA0L and REA0L keep track of start and end points of blocks.
- BRC1 and BRS1 are used to repeat blocks of instructions.
- RSA0 and RSA1 are the start address registers for block repeats.

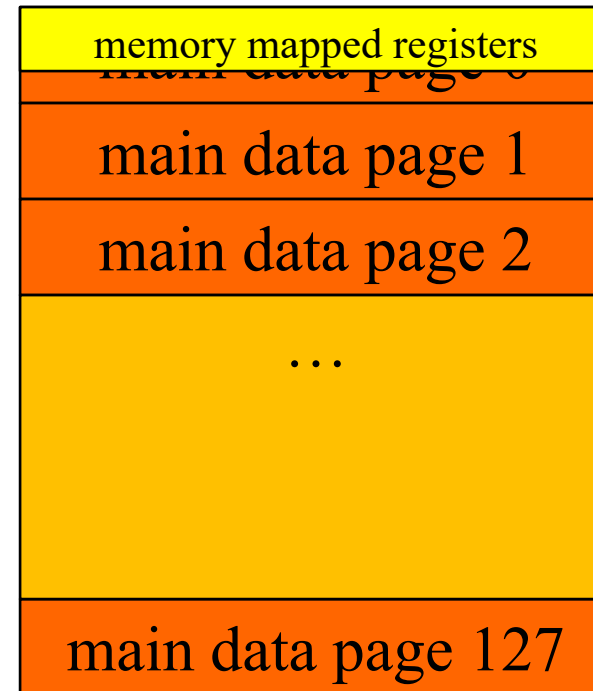
C55x addressing mode and interrupt registers



- DP and DPH set base address for data access.
- PDP determines base address for I/O.
- IER0 and IER1 are interrupt mask registers.
- IFR0 and IFR1 keep track of currently pending registers.
- DBIER0 and DBIER1 are for debugging.

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 direct addressing

- DP addressing accesses data pages:
 - $A_{DP} = DPH[22:15](DP + Doffset)$
- SP addressing accesses stack values:
 - $A_{SP} = SPH[22:15](SP + Soffset)$
- Register-bit direct addressing accesses bits in registers.
- PDP addressing accesses I/O pages:
 - $A_{PDP} = PDP[15:6]PDPoffset$

C55x indirect addressing



- AR indirect addressing uses auxiliary register to point to data.
- Dual AR indirect addressing allows two simultaneous accesses.
- CDP indirect addressing uses CDP to access coefficients.
- Coefficient indirect addressing is similar to CDP indirect but for instructions with 3 memory operands per cycle.

C55x stack operations



- Two stacks: data and system.
- Three different stack configurations:
 - Dual 16-bit stack with fast return has independent data and system stacks.
 - Dual 16-bit stack with slow return, independent data and system stacks but RETA and CFCT are not used for slow returns.
 - 32-bit stack with slow return, SP and SSP are both modified by the same amount.

C55x data operations

- MOV moves data between registers and memory:
 - MOV src, dst $\text{dst} = \text{dst} + \text{src}$
- Varieties of ADDs:
 - ADD src,dst
 - ADD dual(LMEM),ACx,ACy
- Multiplication:
 - MPY src,dst
 - MAC AC,TX,ACy $\text{ACy} = \text{ACy} + (\text{AC} * \text{TX})$

C55x flow of control



- 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 \leq to test loop counter.
 - Make use of compiler---global optimization, software pipelining.

Single-instruction repeat 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.

C55x interrupts



- Handled using subroutine mechanism.
- Four step handling process:
 - Receive interrupt.
 - Acknowledge interrupt.
 - Prepare for ISR by finishing current instruction, retrieving interrupt vector.
 - Processing the interrupt service routine.
- 32 possible interrupt vectors, 27 priorities.

TI C64x



- High-performance VLIW DSP.
 - Up to eight instructions per cycle.
 - Divided into two data paths A and B.
- Floating-point and integer arithmetic
- Harvard architecture.
- Instruction grouping:
 - Fetch packet is a group of instructions executed together.
 - Execute packet instructions are executed together.



Laptop-like performance

TI OMAP3530

- 600 MHz superscaler ARM® Cortex™-A8
- More than 1200 Dhrystone MIPS
- Up to 10 Million polygons per sec graphics
- HD video capable C64x+™ DSP core

Memory

- 128MB LPDDR RAM
- 256MB NAND flash

← 3" →



Flexible expansion

I²C, I²S, SPI,
MMC/SD

DVI-D

JTAG

S-Video

SD/MMC+

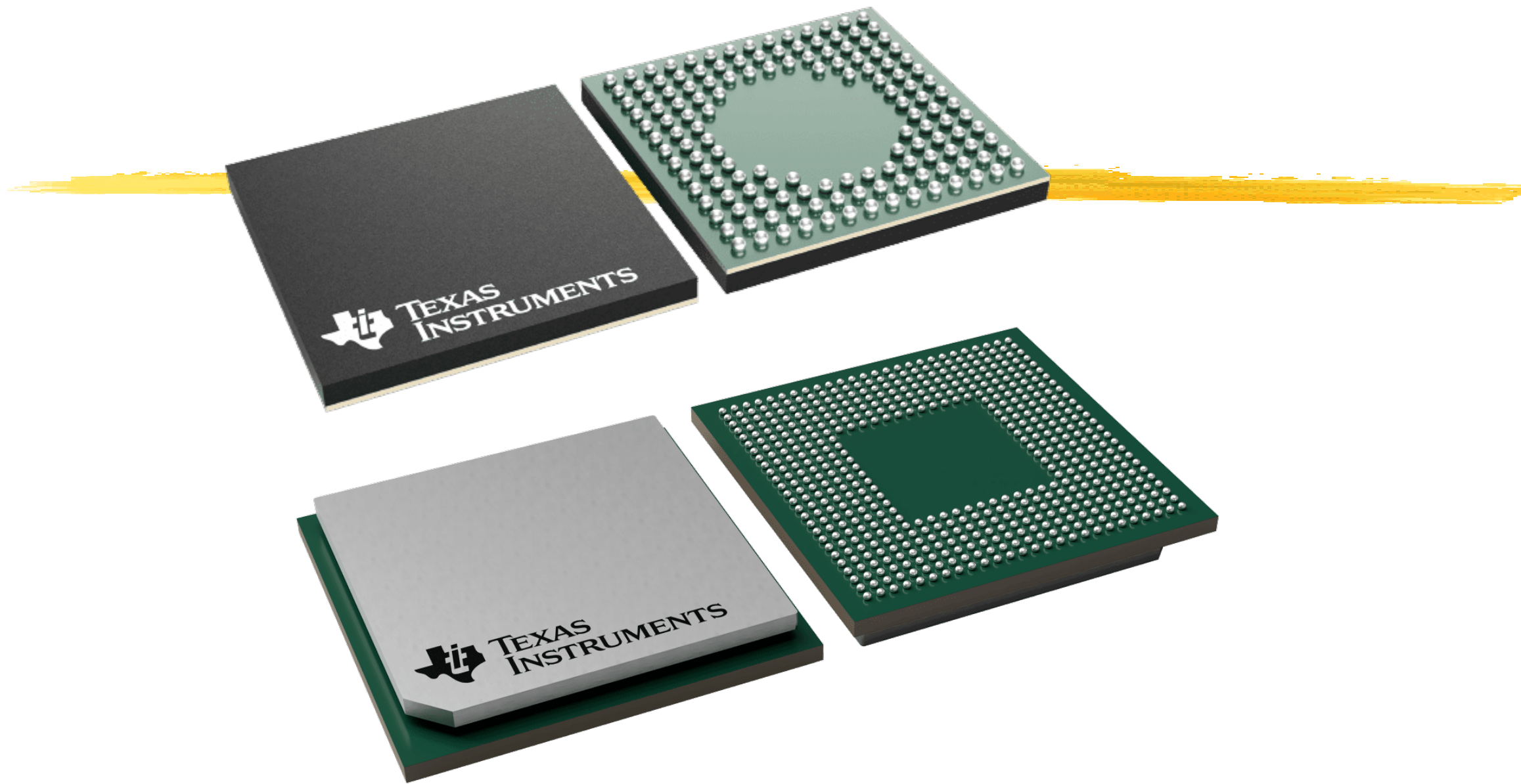
Stereo Out

Stereo In

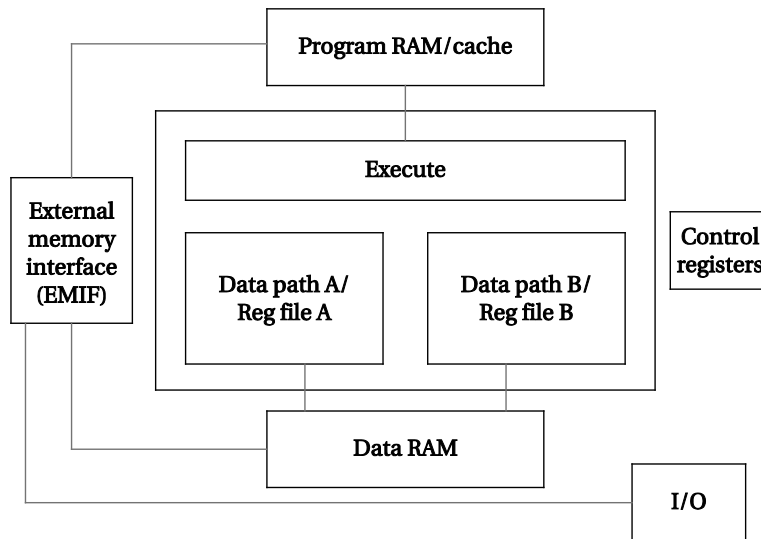
USB 2.0 HS OTG

Alternate Power

RS-232 Serial



C64x block diagram



- Execution divided between two data paths.
- Each data path has its own register file.
- Data must be explicitly copied from one register file to the other.

Function units



- .L1 and .L2 perform 32/40-bit arithmetic, 32-bit logical, data packing/unpacking.
- .S1 and .S2 perform 32-bit arithmetic, 32/40-bit shift and bit-field ops, 32-bit logical ops, branches, etc.
- .M1 and .M2 perform multiplications, bit interleaving, rotation, Galois field multiplication, etc.
- .D1 and .D2 perform address calculations, loads/stores, etc.

Constraints on execution packets



- ❑ Each instruction must execute on a separate function unit.
- ❑ Most .M write combinations are illegal.
- ❑ Most cases of reading multiple values from opposite register file.
- ❑ Delay cycle added for register read after update from cross-path.
- ❑ Cannot execute two load/store usign dest/source from same reg file.
- ❑ At most four reads of a register.
- ❑ Can't write to register twice.

Other features



- Atomic operations for semaphores.
- Circular addressing modes.
- Delay slots to use instruction slots after a multi-cycle operation.
- Interrupts mediated by registers IFR and IER.