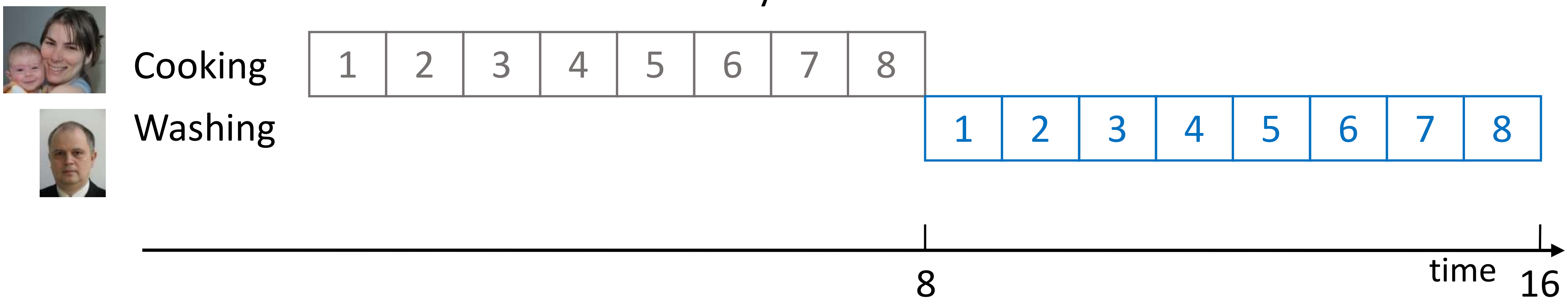
System Performance

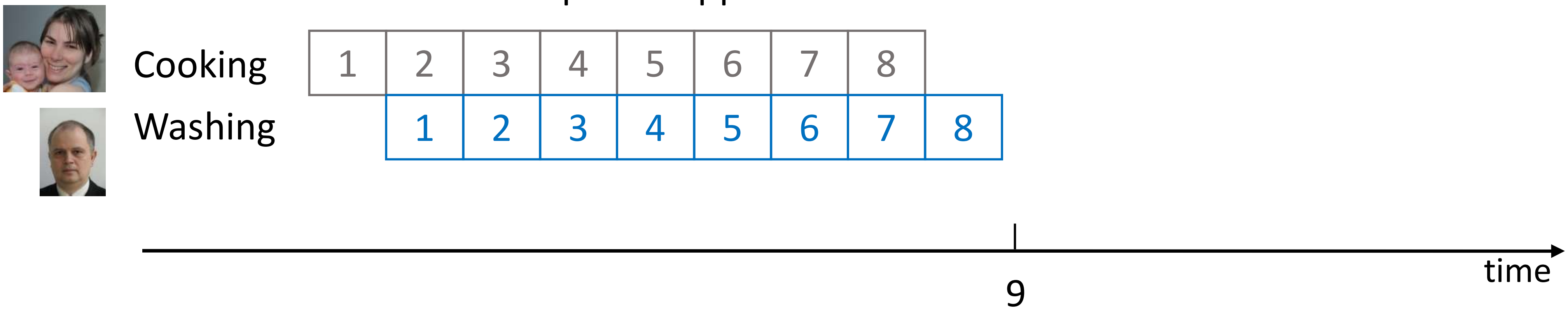
- Pipelining
- Caching & Virtual Memory
- CPU Architectures:
  - CISC & RISC
  - DSP
  - FPU;
- examples (Pentium, Itanium, Power-PC, Strong-ARM)
- VLSI systems:
  - Microcontroller
  - ASIC
  - ASSP

Pipelining - principle

Traditional Style



Pipeline Approach



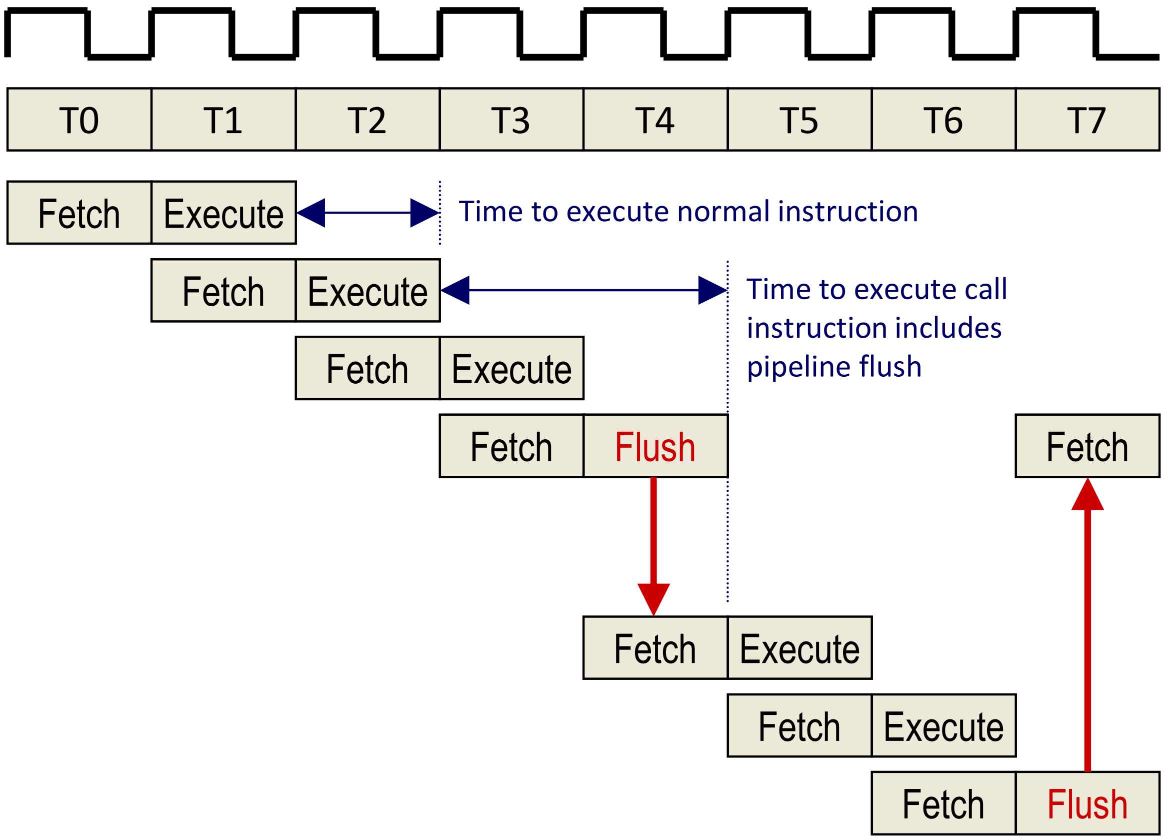
Instruction Pipelining

- Instruction fetch is overlaped with execution of previously fetched instruction

Example Program

```
MAIN  movlw   0x05
      movwf   REG1
      call    SUB1
      addwf   REG2
         ○
         ○
         ○
SUB1  movf    PORTB,w
      return
SUB2  movf    PORTC,w
      return
```

Instruction Cycles



T0	T1	T2	T3	T4	T5	T6	T7
Fetch	Execute						
	Fetch	Execute					
		Fetch	Execute				
			Fetch	Flush			Fetch
				Fetch	Execute		
					Fetch	Execute	
						Fetch	Flush

Time to execute normal instruction

Time to execute call instruction includes pipeline flush

Instruction Pipelining

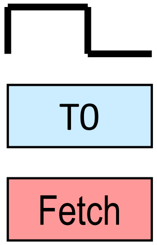
Pre-Fetched Instruction

movlw 0x05

Executing Instruction

–

Instruction Cycles



T0

Fetch

Example Program

```
1   MAIN  movlw   0x05
2         movwf   REG1
3         call    SUB1
4         addwf   REG2
               ⋮
51  SUB1  movf    PORTB,w
52        return
53  SUB2  movf    PORTC,w
54        return
```

Instruction Pipelining

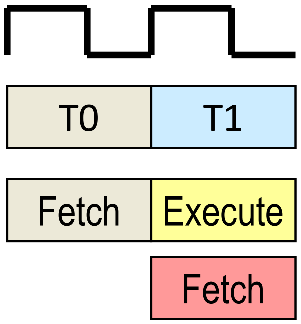
Pre-Fetched Instruction

movwf REG1

Executing Instruction

movlw 0x05

Instruction Cycles



T0	T1
Fetch	Execute
	Fetch

Example Program

```
1   MAIN  movlw   0x05
2         movwf   REG1
3         call    SUB1
4         addwf   REG2
              ⋮
51  SUB1  movf    PORTB,w
52        return
53  SUB2  movf    PORTC,w
54        return
```

Instruction Pipelining

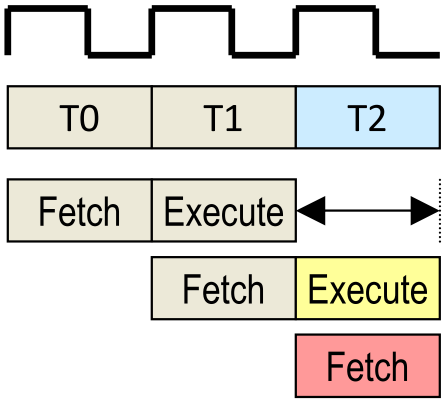
Pre-Fetched Instruction

call SUB1

Executing Instruction

movwf REG1

Instruction Cycles



Time to execute normal instruction

Example Program

```
1   MAIN  movlw   0x05
2         movwf   REG1
3         call    SUB1
4         addwf   REG2
             ⋮
51  SUB1  movf    PORTB,w
52        return
53  SUB2  movf    PORTC,w
54        return
```

Instruction Pipelining

Pre-Fetched Instruction

addwf REG2

Executing Instruction

call SUB1

Instruction Cycles

T0	T1	T2	T3
Fetch	Execute		
	Fetch	Execute	
		Fetch	Execute
			Fetch

Example Program

```
1   MAIN   movlw   0x05
2          movwf   REG1
3          call    SUB1
4          addwf   REG2
                 ⋮
51  SUB1   movf    PORTB,w
52         return
53  SUB2   movf    PORTC,w
54         return
```

Instruction Pipelining

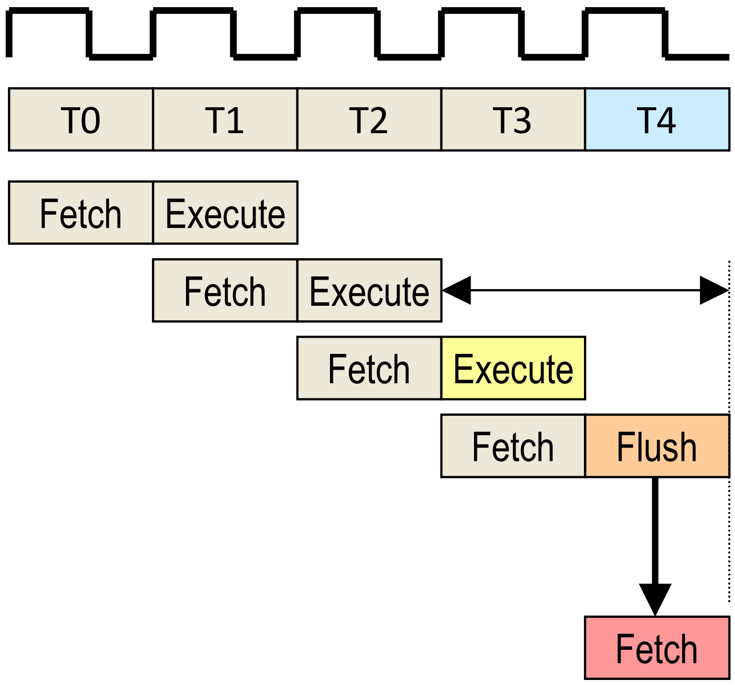
Pre-Fetched Instruction

`movf PORTB,w`

Executing Instruction

`call SUB1`

Instruction Cycles



T0	T1	T2	T3	T4
Fetch	Execute			
	Fetch	Execute		
		Fetch	Execute	
			Fetch	Flush
				Fetch

Time to execute call instruction includes pipeline flush

Example Program

```
MAIN  movlw  0x05
      movwf  REG1
      call   SUB1
      addwf  REG2
         ⋮
SUB1  movf   PORTB,w
      return
SUB2  movf   PORTC,w
      return
```

Instruction Pipelining

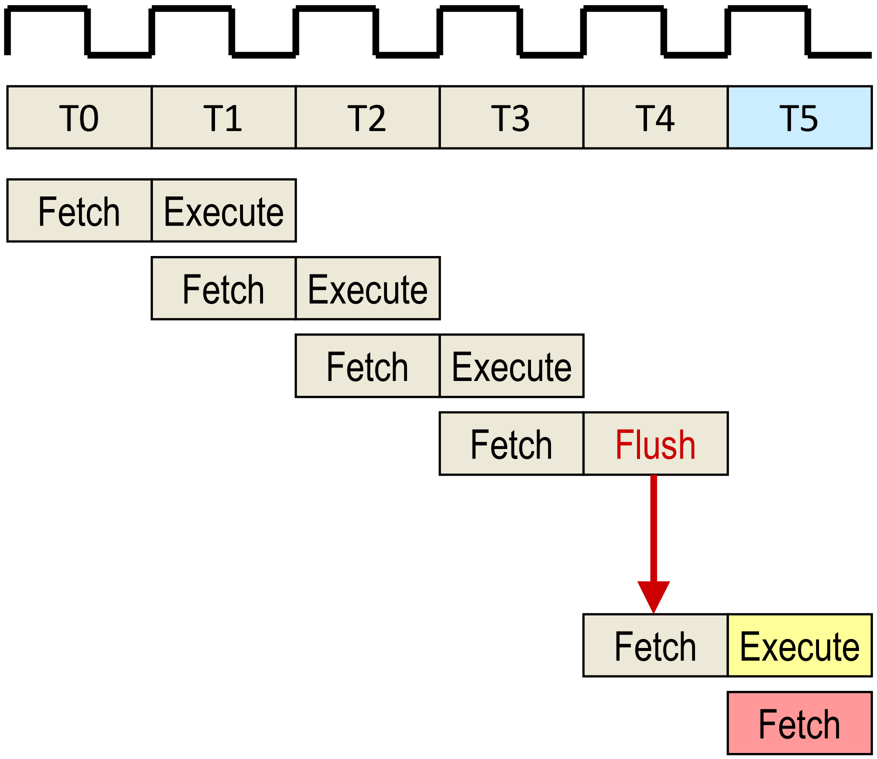
Pre-Fetched Instruction

`return`

Executing Instruction

`movf PORTB,w`

Instruction Cycles



T0	T1	T2	T3	T4	T5
Fetch	Execute				
	Fetch	Execute			
		Fetch	Execute		
			Fetch	Flush	
				Fetch	Execute
					Fetch

Example Program

```
MAIN  movlw   0x05
      movwf   REG1
      call    SUB1
      addwf   REG2
          ⋮
SUB1  movf    PORTB,w
      return
SUB2  movf    PORTC,w
      return
```

Instruction Pipelining

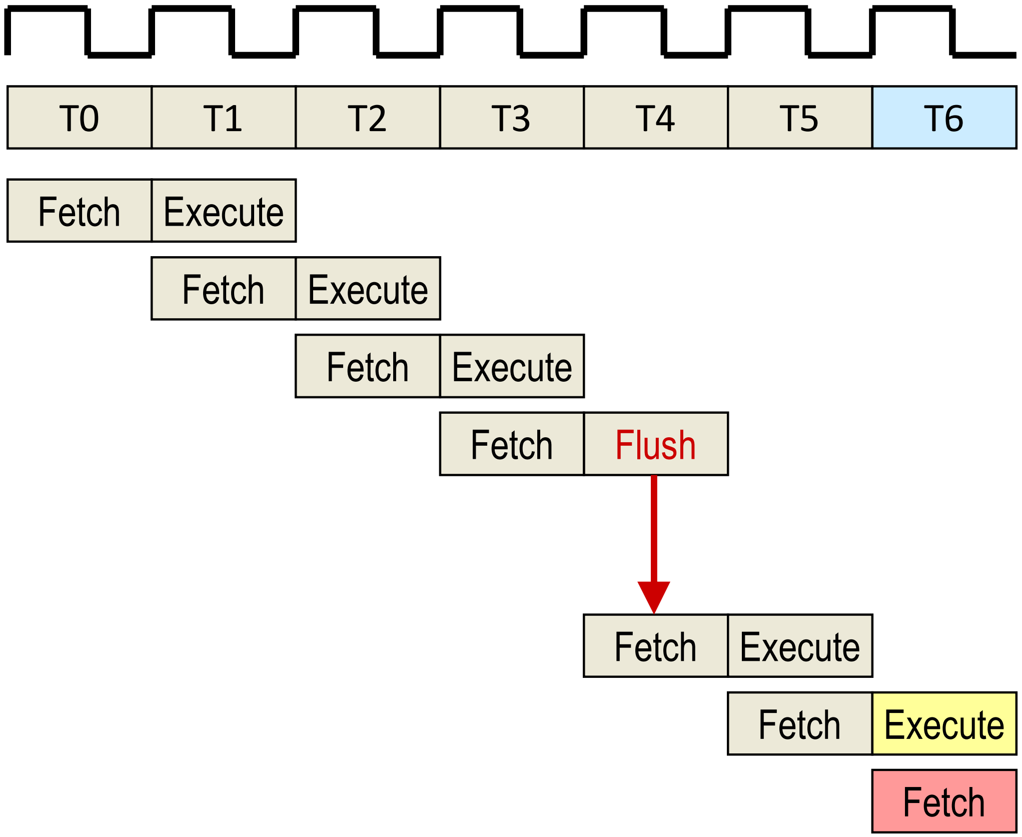
Pre-Fetched Instruction

movf PORTC,w

Executing Instruction

return

Instruction Cycles



Example Program

```
1   MAIN  movlw   0x05
2         movwf   REG1
3         call    SUB1
4         addwf   REG2
              ⋮
51  SUB1  movf    PORTB,w
52        return
53  SUB2  movf    PORTC,w
54        return
```

Instruction Pipelining

Pre-Fetched Instruction

addwf REG2

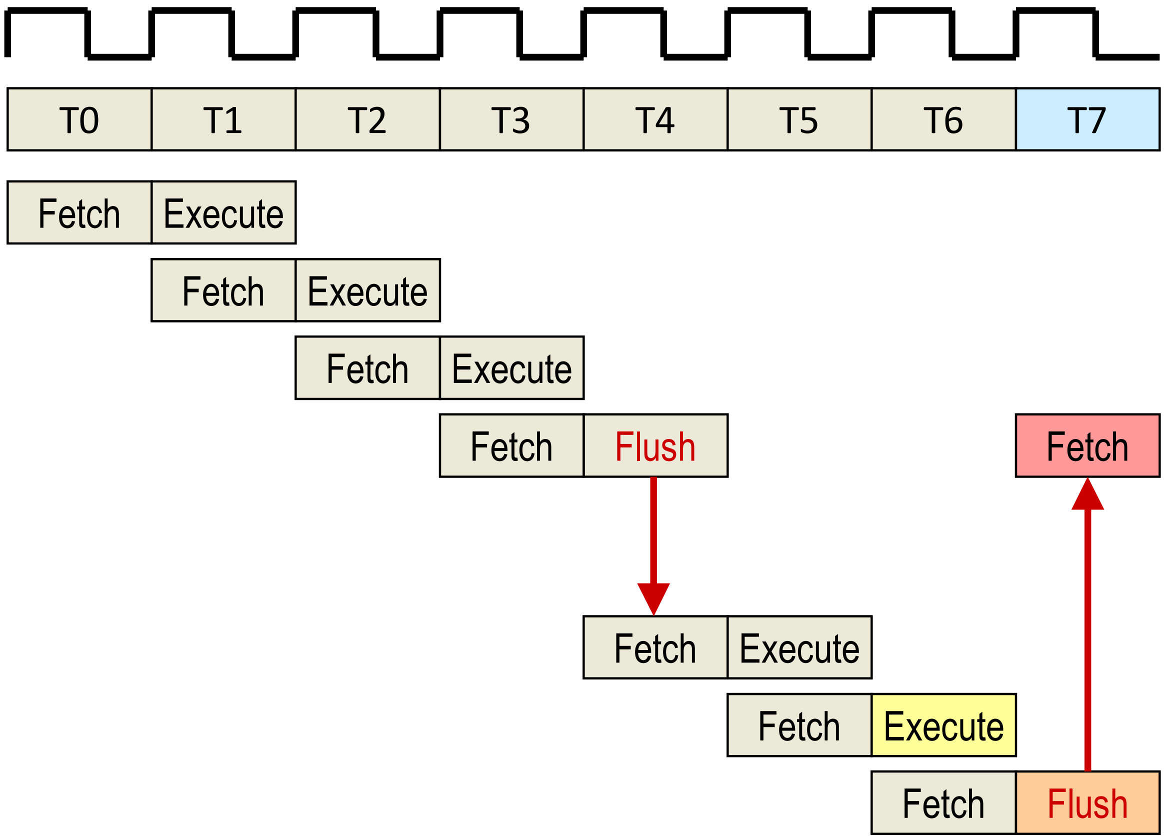
Executing Instruction

return

Example Program

```
1   MAIN   movlw   0x05
2          movwf   REG1
3          call    SUB1
4          addwf   REG2
          ⋮
51  SUB1   movf    PORTB,w
52         return
53  SUB2   movf    PORTC,w
54         return
```

Instruction Cycles



T0	T1	T2	T3	T4	T5	T6	T7
Fetch	Execute						
	Fetch	Execute					
		Fetch	Execute				
			Fetch	Flush			Fetch
				Fetch	Execute		
					Fetch	Execute	
						Fetch	Flush

Multitasking

- Multitasking OS execute multiple programs at the same time
    - by assigning each program a certain percentage of the microprocessor’s time
    - periodically changing which instruction sequence is being executed.
- This is accomplished by a periodic timer interrupt that causes the OS kernel to save the state of the microprocessor’s registers and then reload the registers with preserved state from a different program.
- Each program runs for a while and is paused, after which execution resumes without the program having any knowledge of having been paused.
- In this respect, the individual programs in a multitasking environment appear to have complete control over the computer, despite sharing the resources with others.
- Such a perspective makes programming for a multitasking OS easier, because the programmer does not have to worry about the infinite permutations of other applications that may be running at any given time.
    - A program can be written as if it is the only application running, and the OS kernel sorts out the run-time responsibilities of making sure that each application gets fair time to run on the microprocessor.

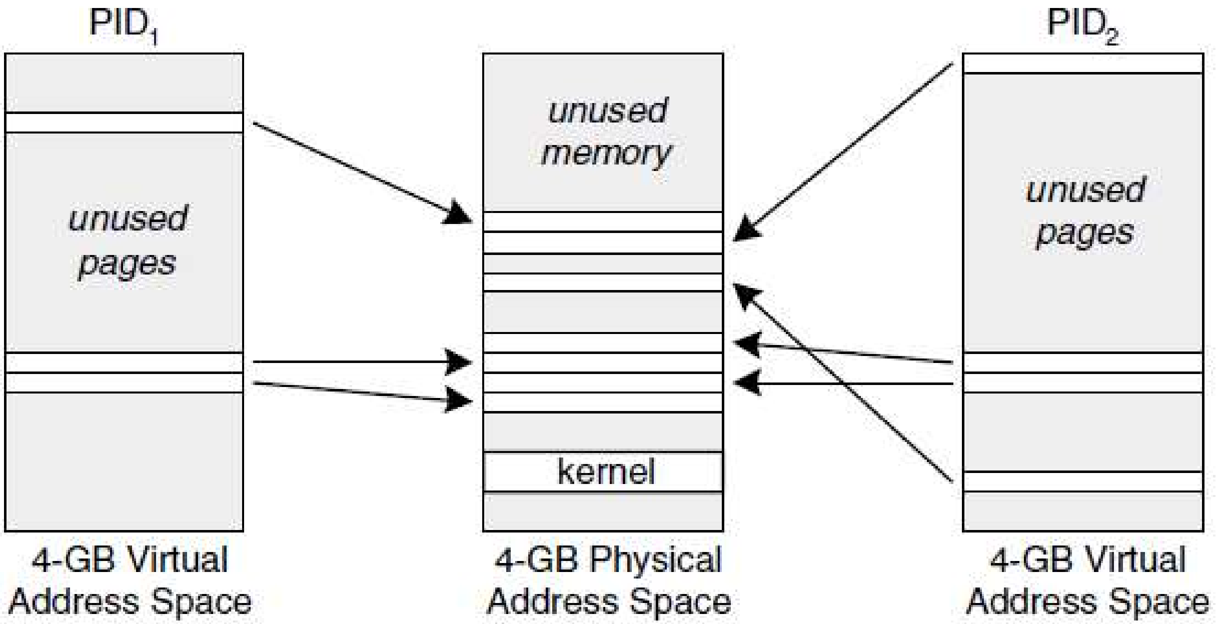
Problems using Multitasking

- conflicts can arise between applications that accidentally modify portions of each other’s memory—either program or data.
- the concern about system-wide fault tolerance.
  - Even if not malicious, programs may have bugs that cause them to crash and write data to random memory locations.
- In such an instance, one errant application could bring down others or even crash the OS if it overwrites program and data regions that belong to the OS kernel.
  - The first problem can be addressed with the honor system by requiring each application to dynamically request memory allocations at run time from the kernel.
  - The kernel can then make sure that each application is granted an exclusive region of memory.
  - However, the second problem of errant writes requires a hardware solution that can physically prevent an application from accessing portions of memory that do not belong to it.

Virtual memory

- Virtual memory is a hardware enforced and software configured mechanism
  - that provides each application with its own private memory space that it can use arbitrarily.
  - This virtual memory space can be as large as the microprocessor’s addressing capability—a full 4 GB in the case of a 32-bit microprocessor.
- Because each application has its own exclusive virtual memory space, it can use any portion of that space that is not otherwise restricted by the kernel.
- Virtual memory frees the programmer from having to worry about where other applications may locate their instructions or data, because applications cannot access the virtual memory spaces of others.
  - In fact, operating systems that support virtual memory may simplify the physical structure of programs by specifying a fixed starting address for instructions, the local stack, and data. UNIX is an example of an OS that does this.
- Each application has its instructions, stack, and data at the same virtual addresses, because they have separate virtual memory spaces that are mutually exclusive and, therefore, not subject to conflict.

32-bit virtual memory mapping


PID$_1$
unused pages
4-GB Virtual Address Space
unused memory
kernel
4-GB Physical Address Space
PID$_2$
unused pages
4-GB Virtual Address Space

32-bit virtual memory mapping

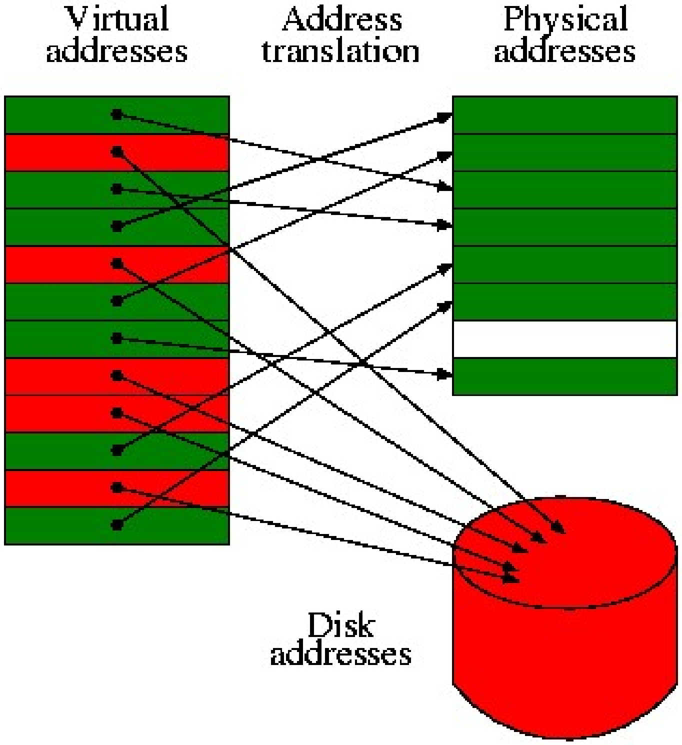
Memory Management Unit

- Clearly, multiple programs cannot place different data at the same address or each simultaneously occupy the microprocessor’s entire address space.
- The OS kernel configures a *hardware memory management unit* (MMU) to map each program’s virtual addresses into unique physical addresses that correspond to actual main memory.
  - Each unique virtual memory space is broken into many small *pages* that are often in the range of 2 to 16 kB in size (4 kB is a common page size).
  - The OS and MMU refer to each virtual memory space with a process ID (PID) field.
  - Virtual memory is handled on a process basis rather than an application basis, because it is possible for an application to consist of multiple semi-independent processes.
- The mapping of virtual memory pages into physical memory is assigned arbitrarily by the OS kernel.
  - The kernel runs in real memory rather than in virtual memory so that it can have direct access to the computer’s physical resources to allocate memory as individual processes are executed and then terminated.

Using a HDD

- Not all mapped virtual memory pages have to be held in physical RAM at the same time.
  - Instead, the total virtual memory allocation on a computer can spill over into a secondary storage medium such as a hard drive.
  - The hard drive will be much slower than DRAM, but not every memory page in every process is used at the same time.
  - When a process is first loaded, its entire instruction image is typically loaded into virtual memory.
  - However, it will take some time for all of those instructions to reach their turn in the execution sequence.
  - During this wait time, the majority of a process's program memory can be stored on the hard drive without incurring a performance penalty. When those instructions are ready to be executed, the OS kernel will have to transfer the data into physical memory.
  - This slows the system down but makes it more flexible without requiring huge quantities of DRAM.
- Part of the kernel's memory management function is to decide which virtual pages should be held in DRAM and which should be *swapped out to the disk.*
- *Pages that have not been used for a* while can be swapped out to make room for new pages that are currently needed.
- If a process subsequently accesses a page that has been moved to the disk, that page can be swapped back into DRAM to replace another page that is not needed at the time.
- A computer with 256 MB of DRAM could, for example, have a 512-MB swap file on its hard drive, enabling processes to share a combined 768 MB of used virtual memory.
- This scheme of expanding virtual memory onto a disk effectively turns the computer's DRAM into a large cache for an even larger disk-based memory.

HDD and the Virtual Memory



Advanced Readings

- Superpipelined and superscalar architectures
  - Mark Balch, COMPLETE DIGITAL DESIGN, A Comprehensive Guide to Digital Electronics and Computer System Architecture, McGRAW-HILL, 2003

Cache

Cache (Level-1 Cache)

- Cache is the level of memory between the CPU and main memory in the memory hierarchy (see Figure).
- Cache can be
  - integrated into a processor
  - or can be off-chip.
- Cache existing on-chip is commonly referred to as level-1 cache, and SRAM memory is usually used as level-1 cache. Because (SRAM) cache memory is typically more expensive due to its speed, processors usually have a small amount of cache, whether on-chip or off-chip.
- Using cache has become popular in response to systems that display a good *locality of reference,* meaning that these systems in a given time period access most of their data from a limited section of memory.
- Cache is used to store subsets of main memory that are used or accessed often.
  - Some processors have one cache for both instructions and data, while other processors have separate on-chip caches for each.

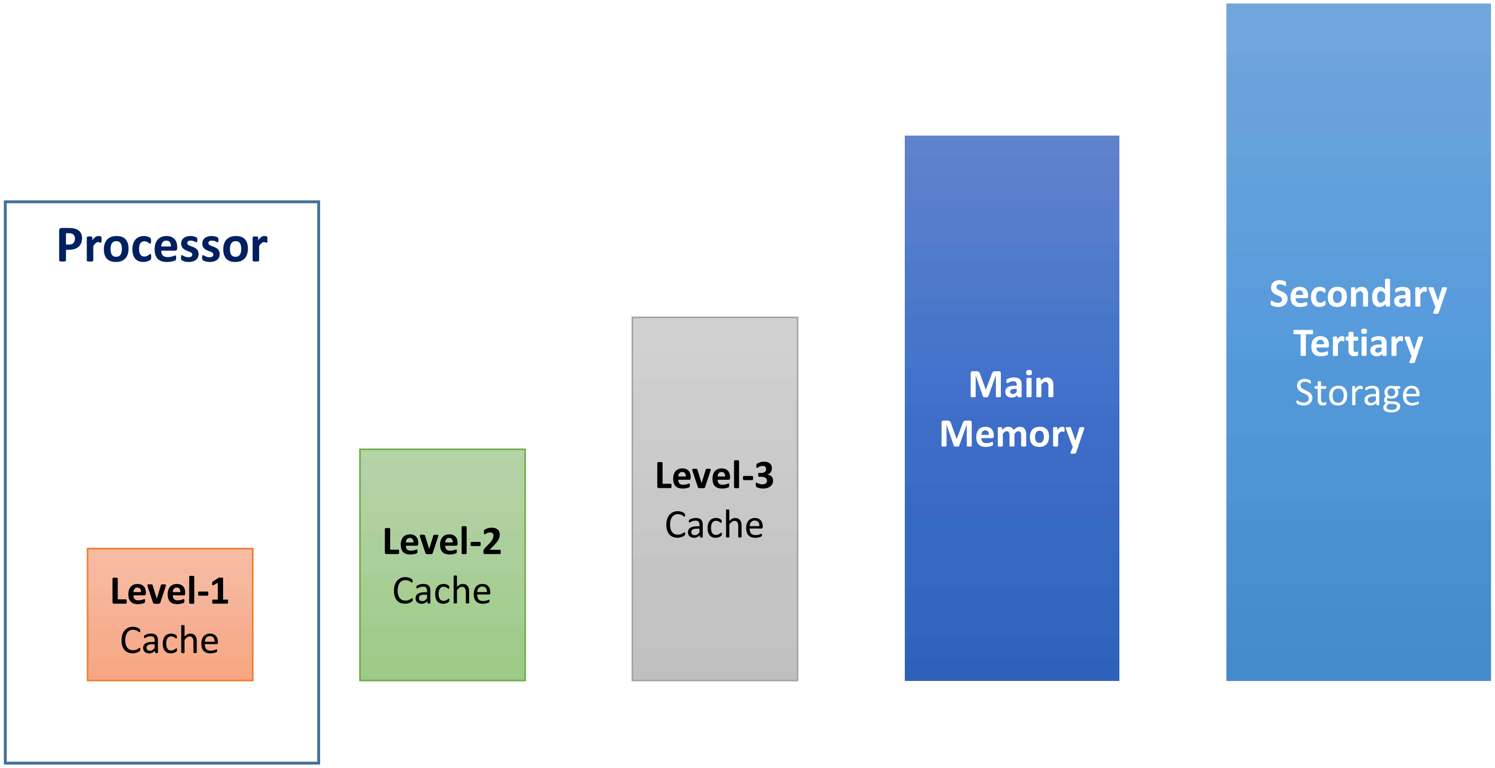


Figure 5: Level-1 cache in the memory hierarchy

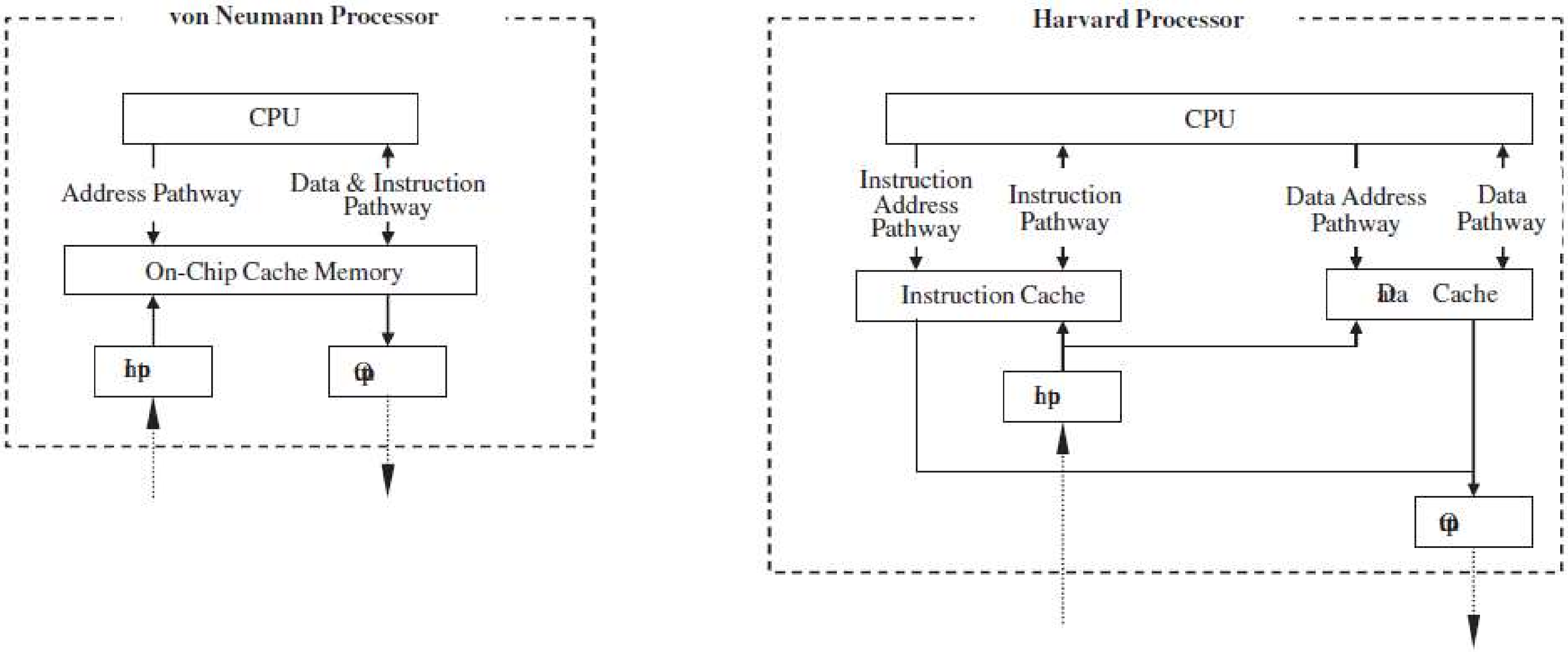


Figure 6: Level-1 cache in the von Neumann and Harvard models

Cache Strategies

- Different strategies are used when writing to and reading data from level-1 cache and main memory.
- These strategies include transferring data between memory and cache in:
  - one-word or
  - multiword blocks*.*
- These blocks are made up of data from main memory, as well as the location of that data in main memory (called **tags*).***

Cache Strategies

- In the case of writing to memory, given some memory address from the CPU, this address is translated to determine its equivalent location in level-1 cache, since cache is a snapshot of a subset of memory.
- Writes must be done in both cache and main memory to ensure that cache and main memory are ***consistent (have the same value).***
- The **two most common write strategies** to guarantee this are
  - **write-through**, in which data is written to both cache and main memory every time, and
  - **write-back**, in which data is initially only written into cache, and only when it is to be bumped and replaced from cache is it written into main memory.

- When the CPU wants to read data from memory, level-1 cache is checked first.
    - If the data is in cache, it is called a **cache hit.**
        - The data is returned to the CPU and the memory access process is complete.
    - If the data is not located in level-1 cache, it is called **cache miss.**
- **Off-chip caches** (if any) are then checked for the data desired.
    - If this is a miss, then main memory is accessed to retrieve and return the data to the CPU.

Cache Data Storage Schemes

- Data is usually stored in cache in one of three schemes:
  - **Direct Mapped**, where data in cache is located by its associated block address in memory (using the "tag" portion of the block).
  - **Set Associative**, where cache is divided into sets into which multiple blocks can be placed. Blocks are located according to an index field that maps into a cache's particular set.
  - **Full Associative**, where blocks are placed anywhere in cache, and must be located by searching the entire cache every time.

Direct Mapped Cache Scheme

- In the direct mapped cache scheme, addresses in cache are divided into sections called blocks.
- Every block is made up of the data, a valid tag (flag indicating if block is valid), and a tag indicating the memory address(es) represented by the block.
- In this scheme, data is located by its associated block address in memory, using the “tag” portion of the block.
- The tag is derived from the actual memory address, and is made up of three sections: a tag, an index, and an offset.
- The index value indicates the block, the offset value is the offset of the desired address within the block, and the tag is used to compare with the actual address tag to insure the correct address was located.

Set Associative Cache Scheme

- The set associative cache scheme is one in which cache is divided into sections called *sets,* and within each set, multiple blocks are located at the set-level.
- The set associative scheme is implemented at the set-level.
- At the block level, the direct-mapped scheme is used.
- Essentially, all sets are checked for the desired address via a universal broadcast request.
- The desired block is then located according to a tag that maps into a cache’s particular set.

Full Associative Cache Scheme

- The full associative cache scheme, like the set associative cache scheme, is also composed of blocks.
- In the full associative scheme,
  - blocks are placed anywhere in cache, and
  - must be located by searching the entire cache every time.

Full Associative Cache Scheme

- As with any scheme, each of the cache schemes has its **strengths** and **drawbacks**.
- Whereas the set associative and full associative schemes are slower than the direct mapped,
- the direct mapped cache scheme runs into performance problems when the block sizes get too big.
- On the flip side, the cache and full associative schemes are less predictable than the direct mapped cache scheme, since their algorithms are more complex.

Common Cache Selection And Replacement Schemes

- The actual cache swapping scheme is determined by the architecture.
- The most common cache selection and replacement schemes include:
  - **Optimal, using future reference time,** swapping out pages that won't be used in the near future.
  - **Least recently used (LRU)**, which swaps out pages that were used the least recently.
  - **FIFO (first in, first out)** is another scheme which, as its name implies, swaps out the pages that are the oldest, regardless of how often they are accessed in the system. While a simpler algorithm then LRU, FIFO is much less efficient.
  - **Not recently used (NRU)**, swaps out pages that were not used within a certain time period.
  - **Second chance, FIFO scheme** with a reference bit, if "0" will be swapped out (a reference bit is set to "1" when access occurs, and reset to "0" after the check).
  - **Clock paging,** pages replaced according to clock (how long they have been in memory), in clock order, if they haven't been accessed (a reference bit is set to "1" when access occurs, and reset to "0" after the check).

Level-2+ Caches

- Level 2+ (level 2 and higher) cache is the level of memory that exists between the CPU and main memory in the memory hierarchy.
- cache that is external to the processor is introduced, which is caches higher than level 1.
- SRAM memory is usually used as external cache (like level-1 cache), because the purpose of cache is to improve the performance of the memory system, and SRAM is faster than DRAM. Since (SRAM) cache memory is typically more expensive because of its speed, processors will usually have a small amount of cache (on-chip, off-chip, or both).
- Using cache became popular in response to systems that displayed a good locality of reference, meaning that these systems, in a given time period, accessed most of their data from a limited section of memory.
- Basically, cache is used to store subsets of main memory that are used or accessed often, capitalizing on the locality of reference and making main memory seem to execute faster. Because cache holds copies of what is in main memory, it gives the illusion to the master processor that it is operating from main memory even if actually operating from cache.

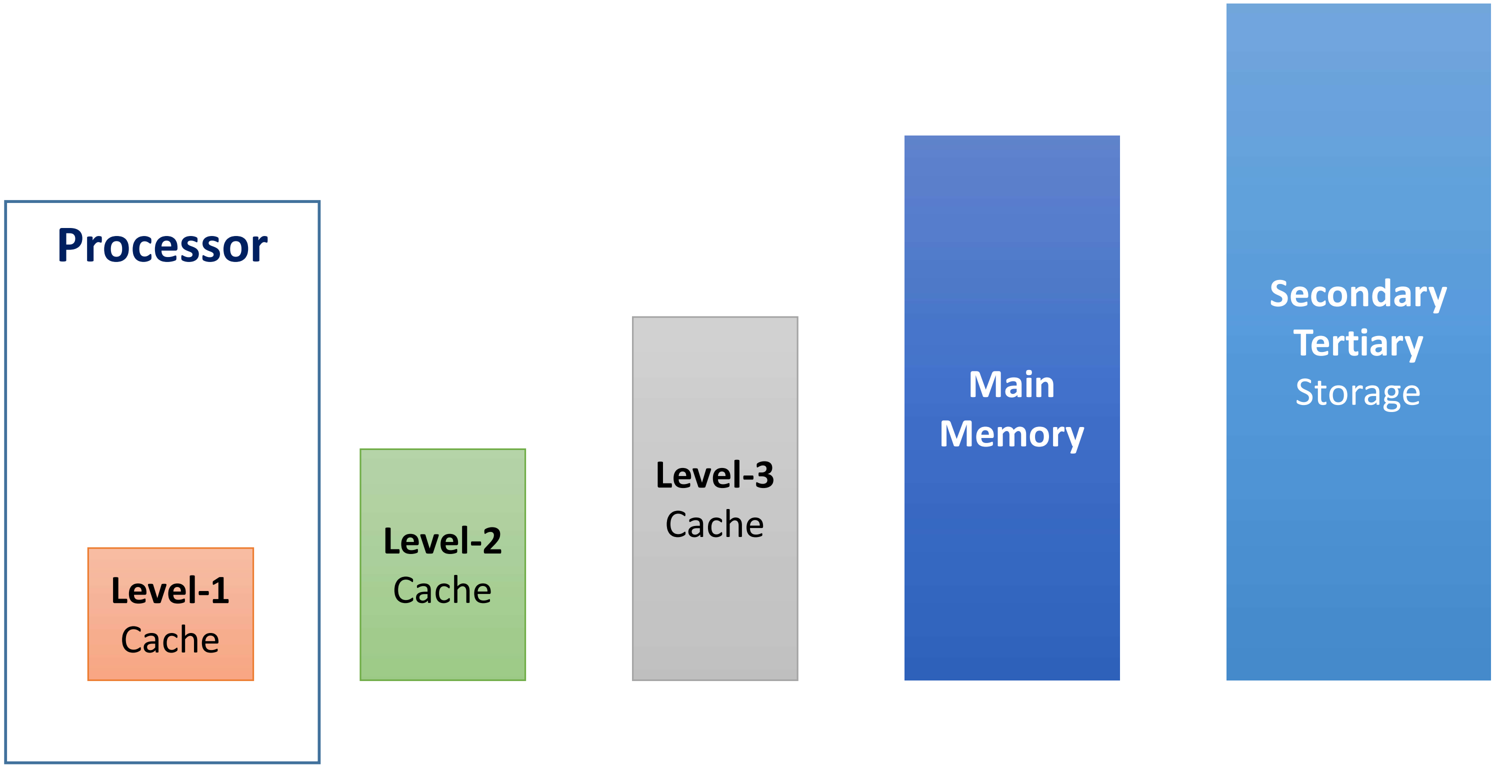
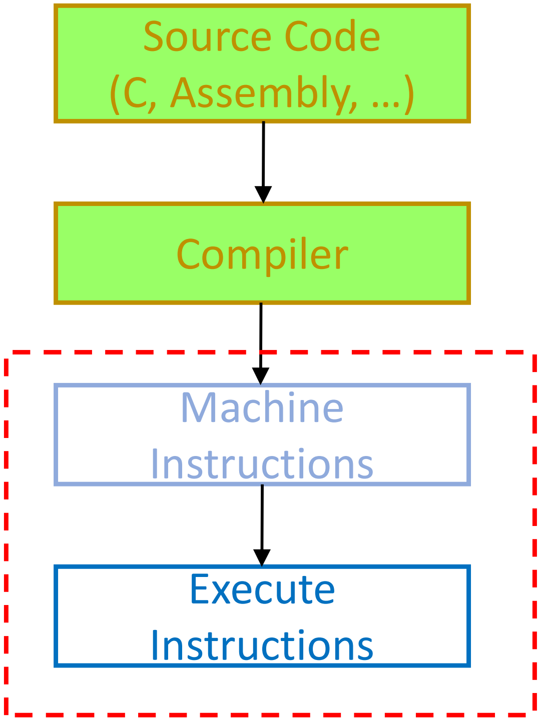


Figure 5-1: Memory hierarchy

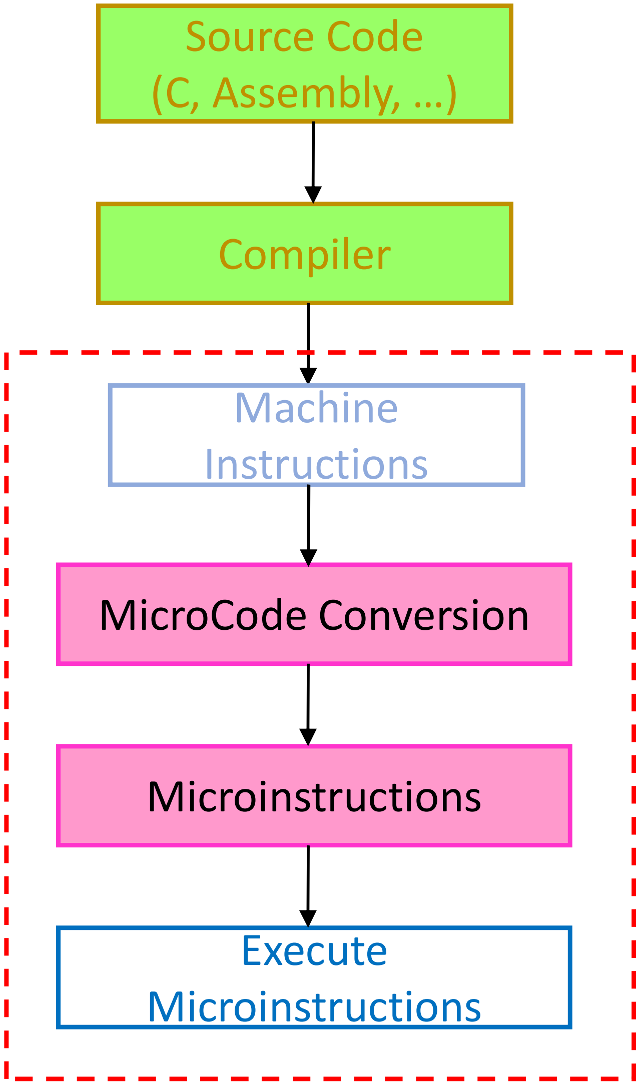
Managing Cache

- In systems with memory management units (MMU) to perform the translation of addresses (see Section 5.4), cache can be integrated between the master processor and the MMU, or the MMU and main memory.
- There are advantages and disadvantages to both methods of cache integration with an MMU, mostly surrounding the handling of DMA devices that allow data to access off-chip main memory directly without going through the main processor. (Direct memory access is discussed in Chapter 6, Board I/O.)
- When cache is integrated between the master processor and MMU, only the master processor access to memory affects cache; therefore, DMA writes to memory can make cache inconsistent with main memory unless master processor access to memory is restricted while DMA data is being transferred or cache is being kept updated by other units within the system besides the master processor.
- When cache is integrated between the MMU and main memory, more address translations must be done, since cache is affected by both the master processor and DMA devices.
- In some systems, a memory controller may be used to manage a system with external cache (data requests and writes, for instance).

RISC vs CISC

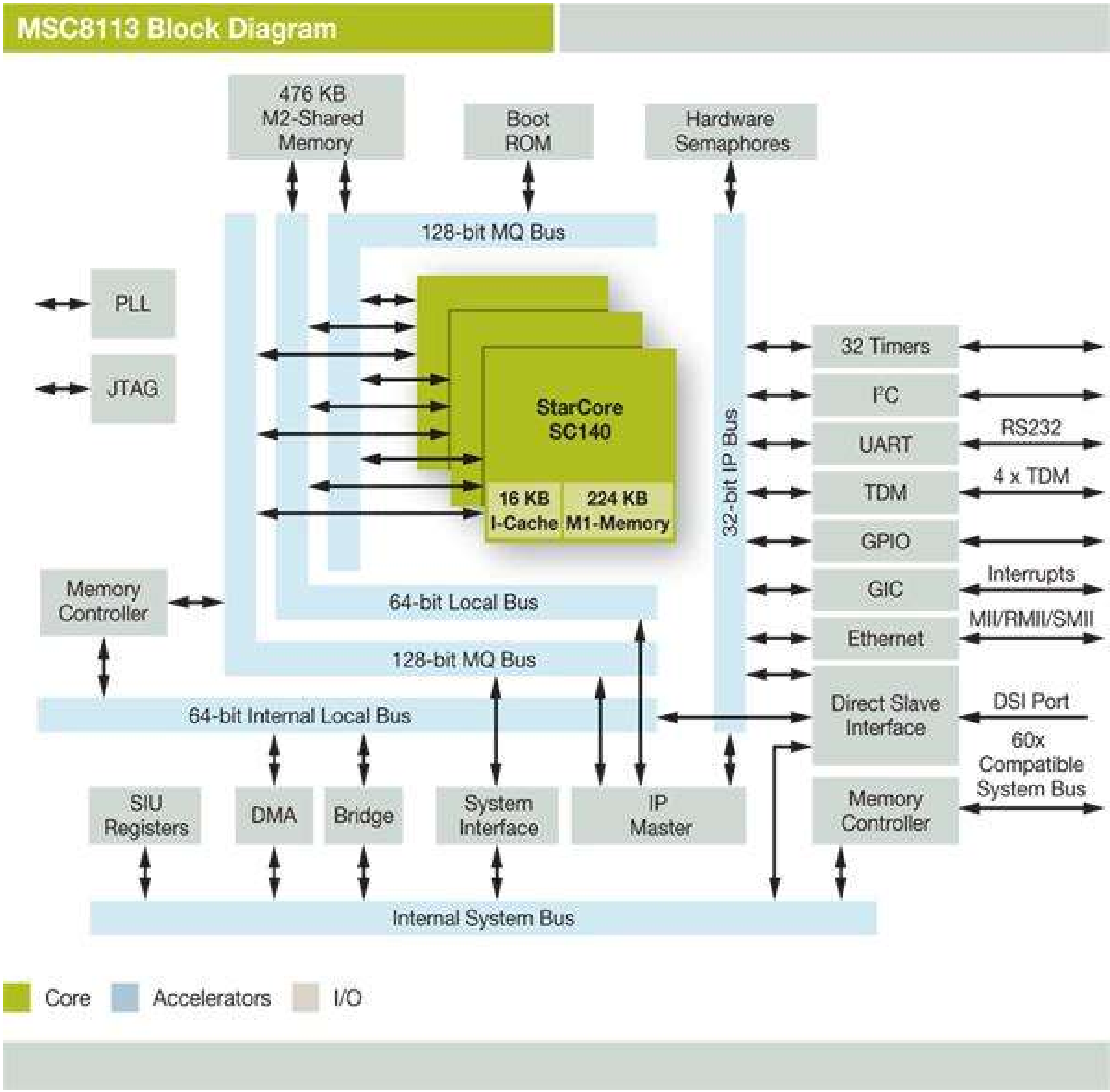


RISC type CPU

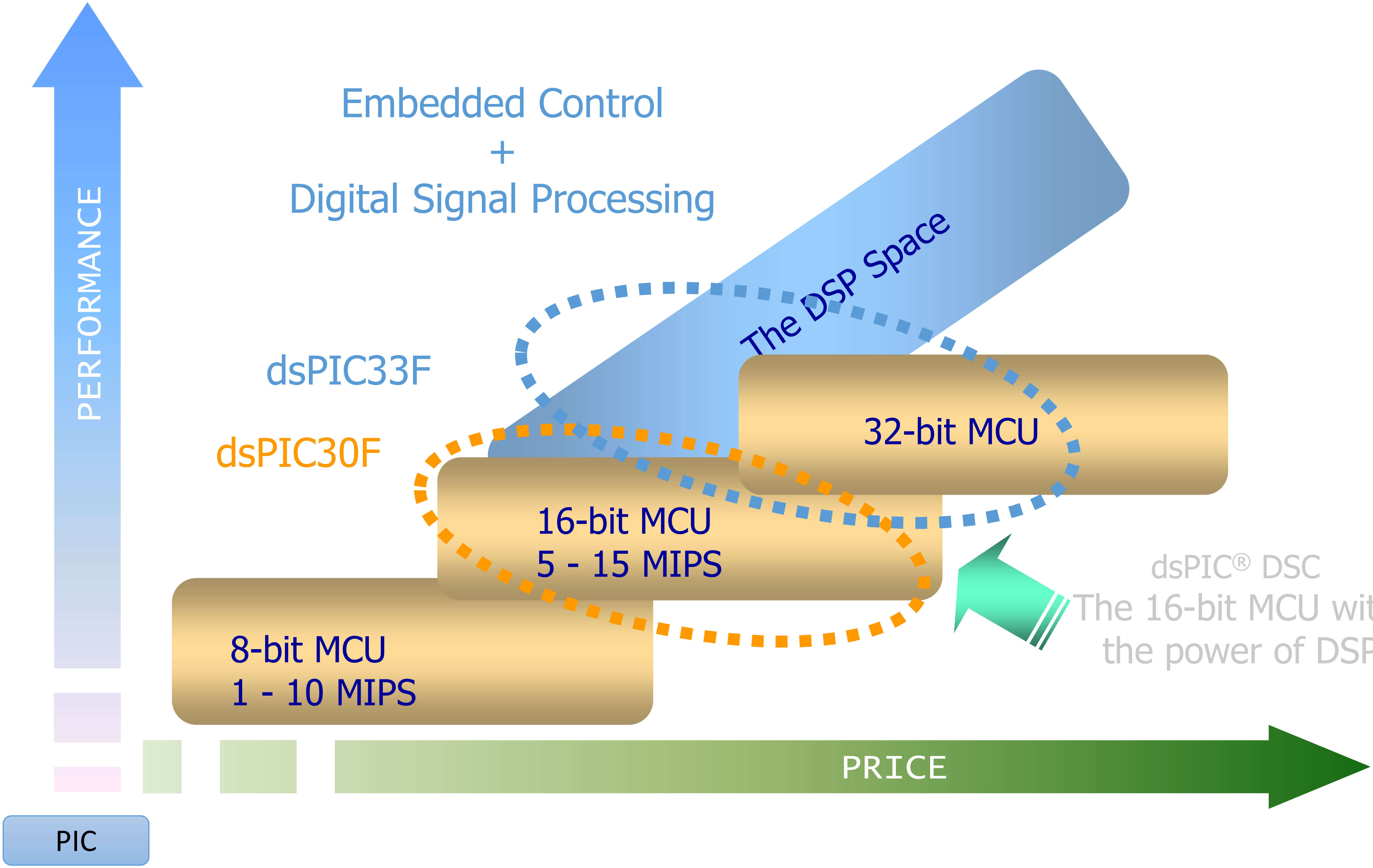


CISC type CPU

DSP


MSC8113 Block Diagram
476 KB M2-Shared Memory
Boot ROM
Hardware Semaphores
128-bit MQ Bus
PLL
JTAG
StarCore SC140
16 KB I-Cache
224 KB M1-Memory
32-bit IP Bus
32 Timers
I²C
UART
RS232
TDM
4 x TDM
GPIO
GIC
Interrupts
Ethernet
MII/RMII/SMII
Memory Controller
64-bit Local Bus
128-bit MQ Bus
64-bit Internal Local Bus
Direct Slave Interface
DSI Port
60x Compatible System Bus
SIU Registers
DMA
Bridge
System Interface
IP Master
Memory Controller
Internal System Bus
Core
Accelerators
I/O

What is Digital Signal Control?



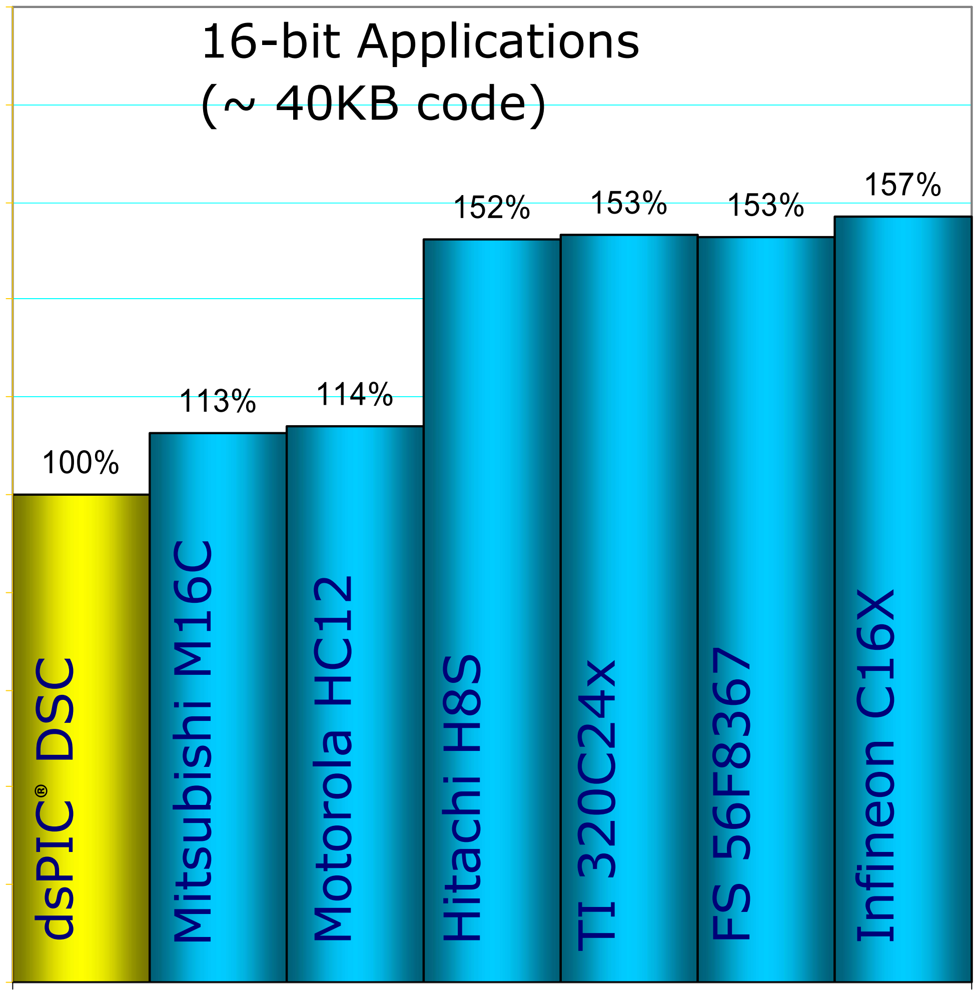
dsPIC® DSC Family: Architected from Scratch

- Seamlessly integrates a DSP and an MCU
- MCU look and feel, easy to use
- Competitive DSP performance
- Optimized for C compiler
- Fast, deterministic, flexible interrupts
- Excellent RTOS support

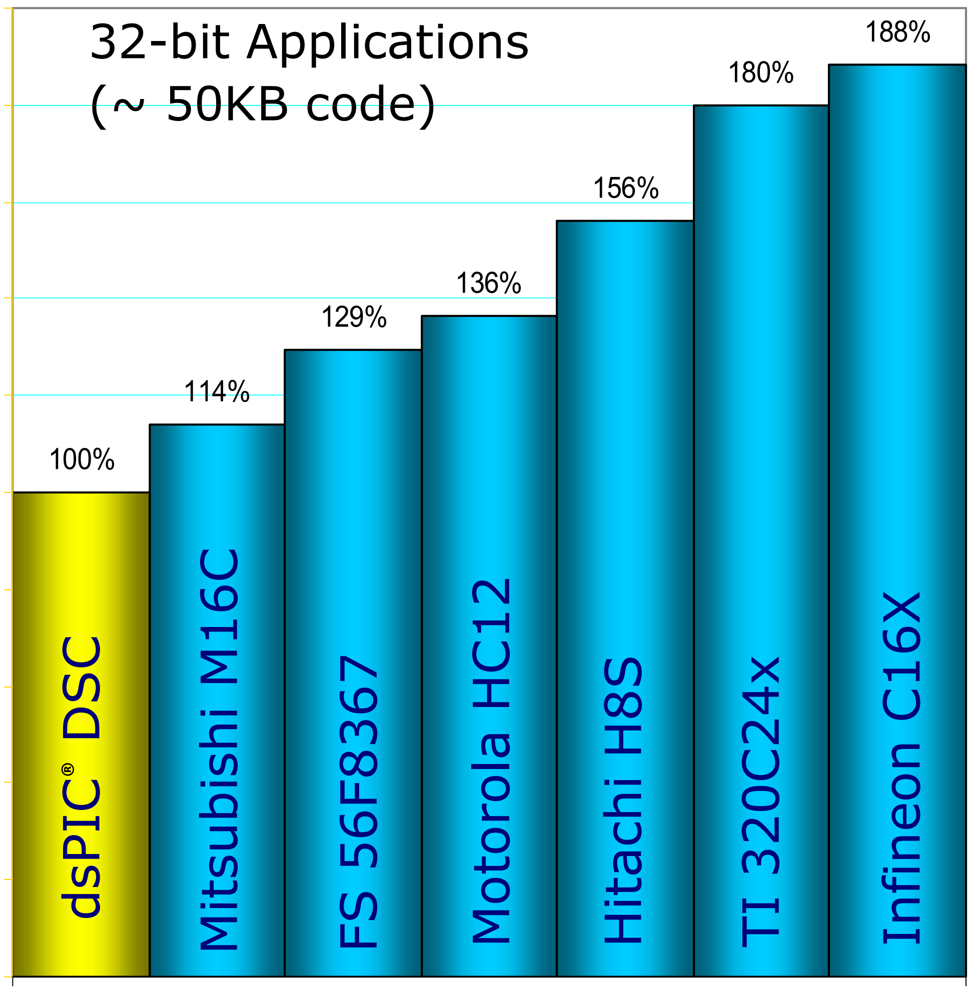
PIC

dsPIC® DSC Highly Optimized C compiler Control Centric Benchmarks

Relative Code Size



MPLAB® C30 v1.30



MPLAB® C30 v1.30

PIC

Other CPUs

dsPIC® DSC Architecture

- Main Features
  - Tightly Integrated Core
    - Operable as an MCU & a DSP
    - Modified Harvard Architecture
    - 16 x 16-bit working register array
  - Data Memory
    - 16 bits wide
    - Linearly addressable up to 64KB
  - Program Memory
    - 24-bit wide Instructions
    - Linearly addressable up to 12 MB

PIC

dsPIC® DSC Architecture

- Main Features (continued)
  - Many integrated peripherals
  - Software stack
  - Efficient Operation
    - Fast, deterministic interrupt response
    - Three operand instructions: C = A + B
    - Extensive addressing modes
  - DMAC w/ dual port SRAM - 8 channels for peripherals

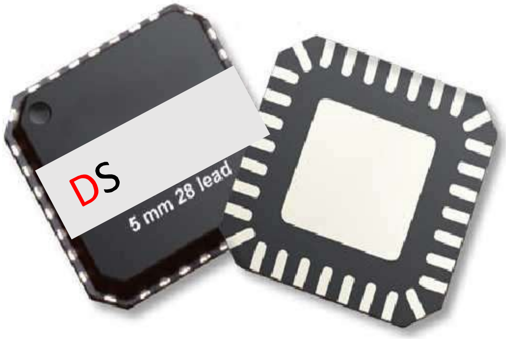
PIC

dsPIC® DSC Operating Parameters

Feature	dsPIC30F	dsPIC33F
□ Operating Speed:	DC to 30 MIPS	DC to 40 MIPS
□ VDD:(VDC)	2.5 to 5.5	3.0 to 3.6
□ Temp:	-40°C to +125°C	-40°C to +85°C
□ Program Memory:	Flash	Flash
□ Data Memory:	SRAM, EEPROM	SRAM, Self-write Flash

□ Package sizes

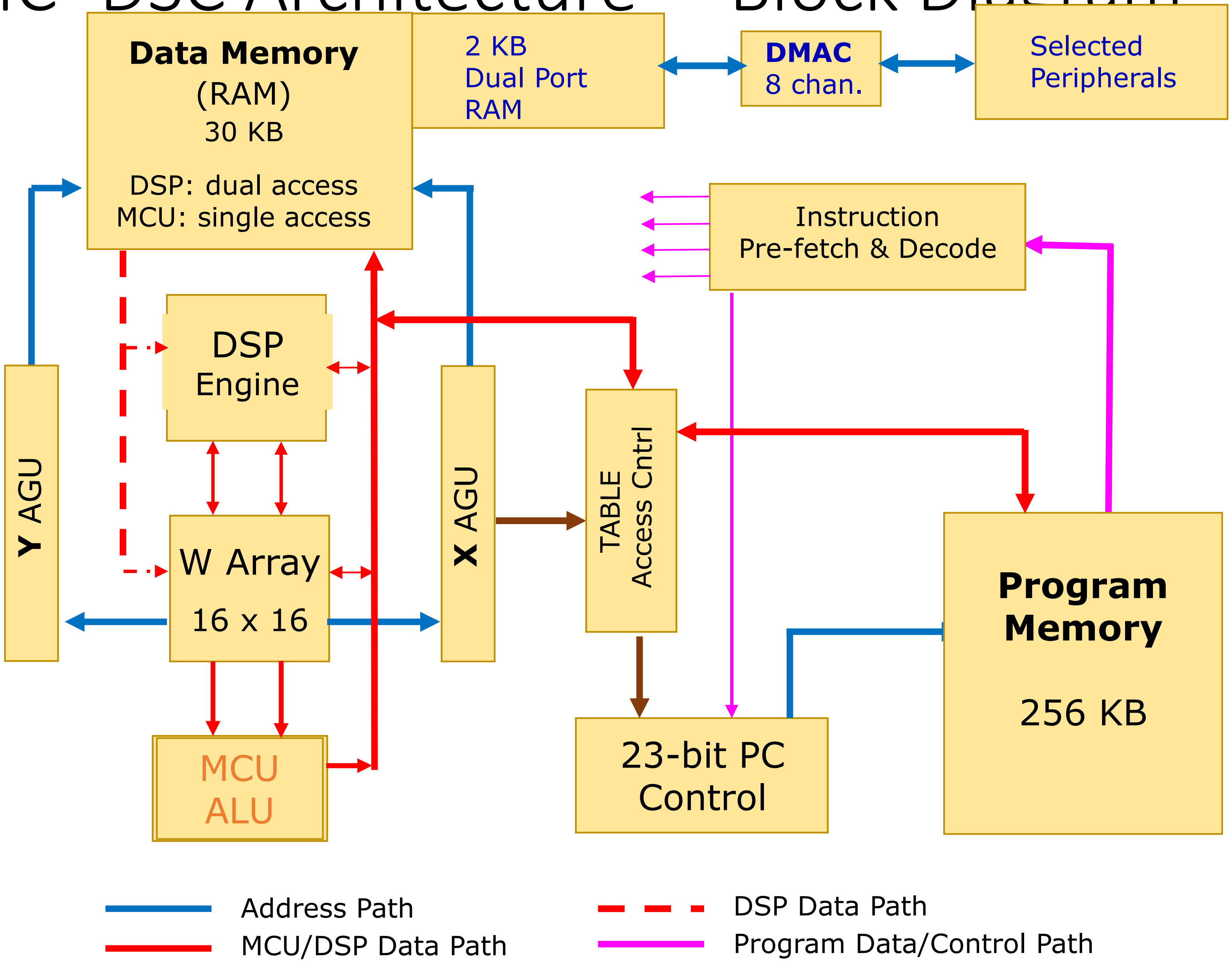
- 18-pin SO & SP
- 28-pin SO, SP and QFN
- 40-pin SP; 44-pin TQFP, QFN
- 64-, 80- and 100- pin TQFP



28 lead QFN: 6 x 6 x 0.9 mm

PIC

dsPIC® DSC Architecture Block Diagram



Data Memory
(RAM)
30 KB

DSP: dual access
MCU: single access

2 KB
Dual Port
RAM

DMAC
8 chan.

Selected
Peripherals

Instruction
Pre-fetch & Decode

DSP
Engine

Y AGU

X AGU

W Array
16 x 16

TABLE
Access Cntrl

MCU
ALU

23-bit PC
Control

Program
Memory

256 KB

Address Path

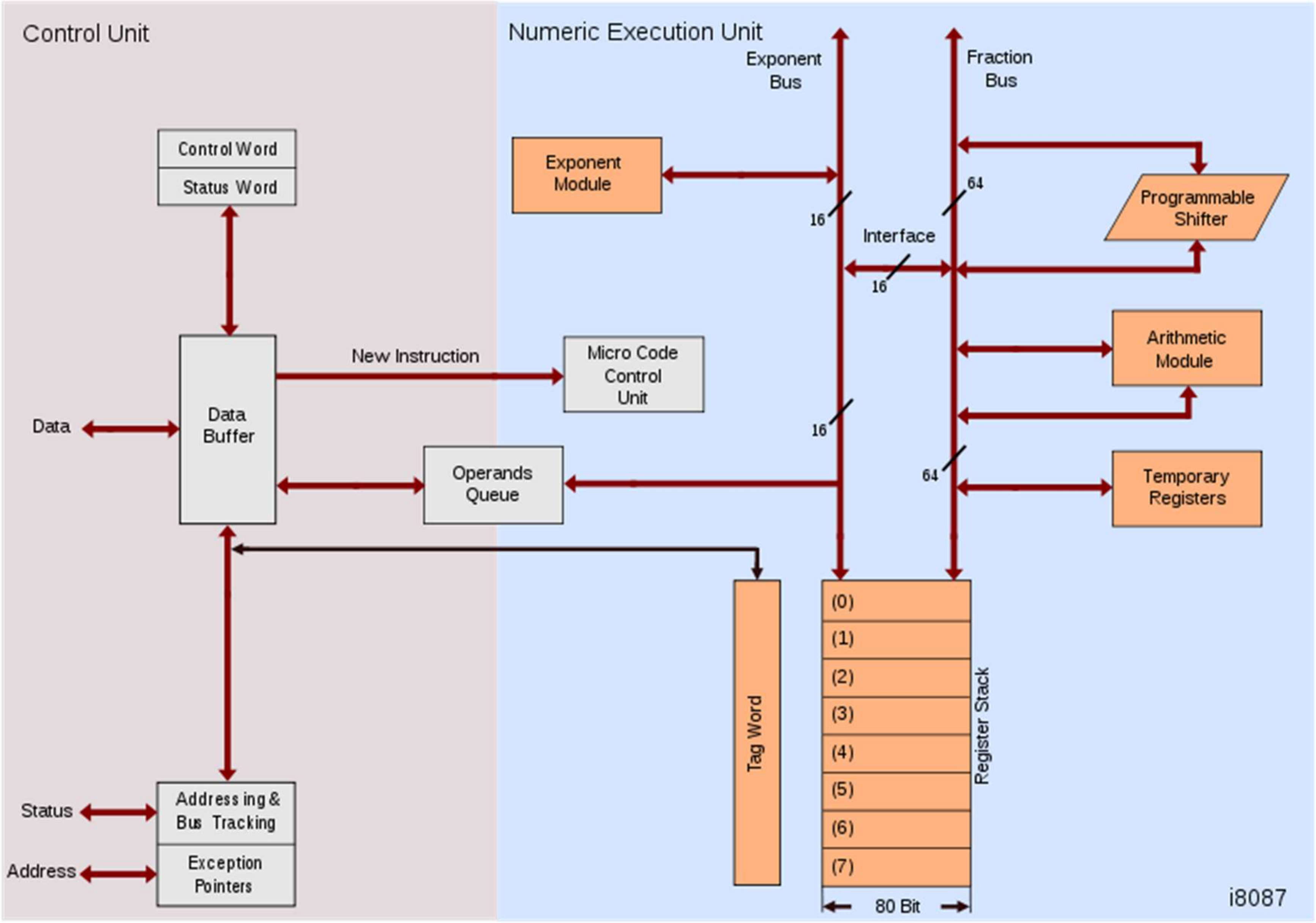
MCU/DSP Data Path

DSP Data Path

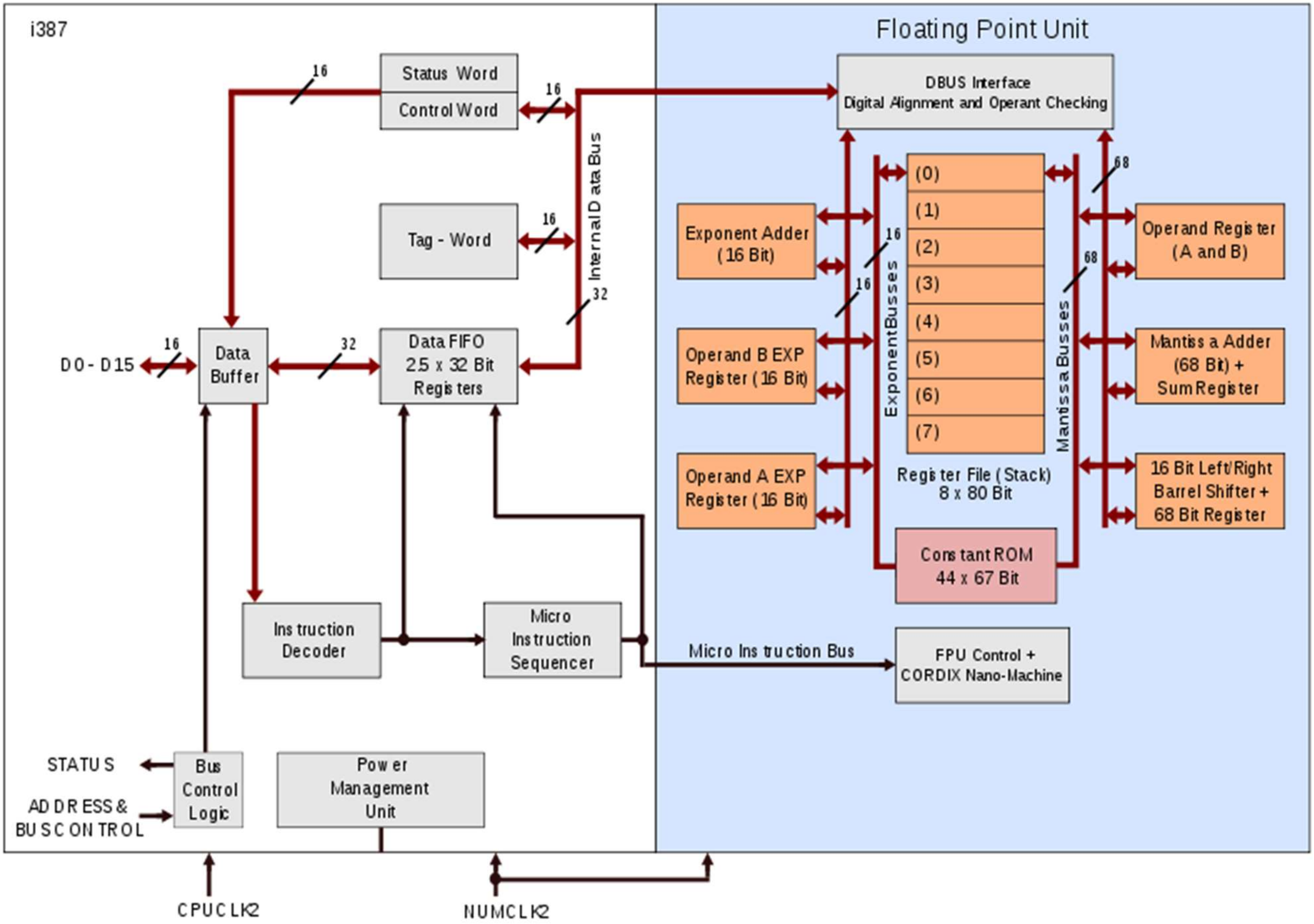
Program Data/Control Path

PIC

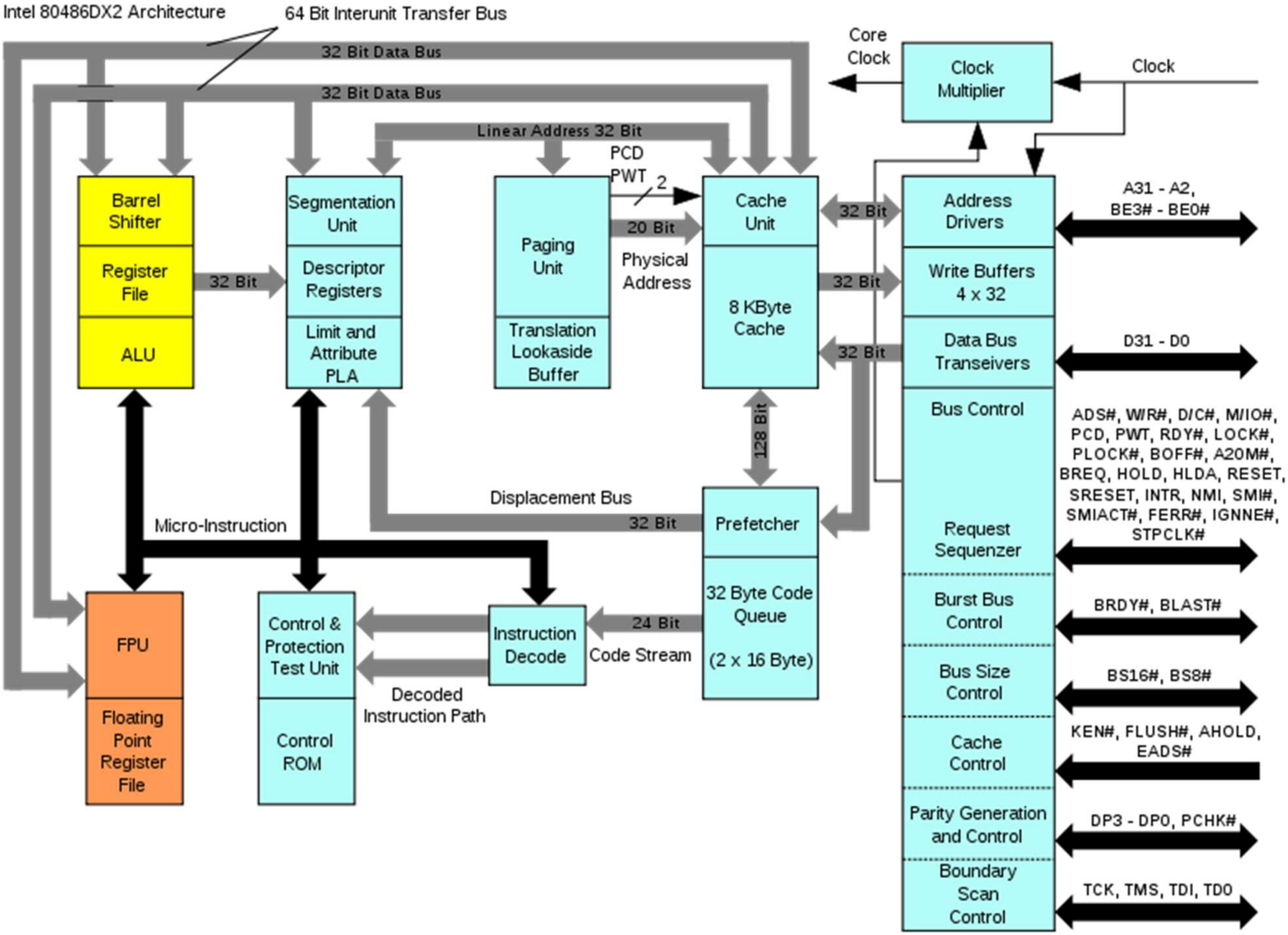
8087


Control Unit
Numeric Execution Unit
Exponent Bus
Fraction Bus
Control Word
Status Word
Exponent Module
16
64
Programmable Shifter
Interface
16
New Instruction
Micro Code Control Unit
Arithmetic Module
Data
Data Buffer
16
64
Operands Queue
Temporary Registers
Tag Word
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
Register Stack
Status
Addressing & Bus Tracking
Address
Exception Pointers
80 Bit
i8087

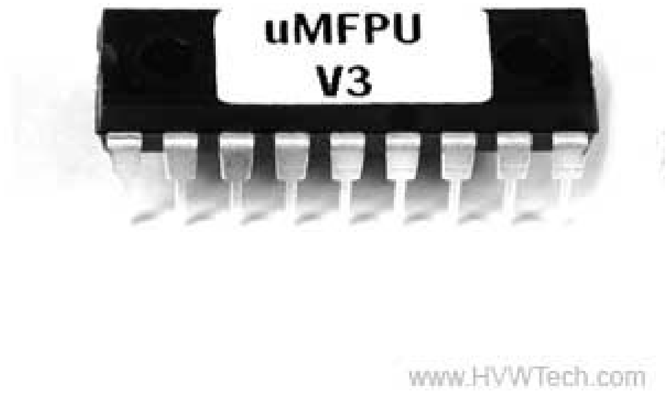
80387

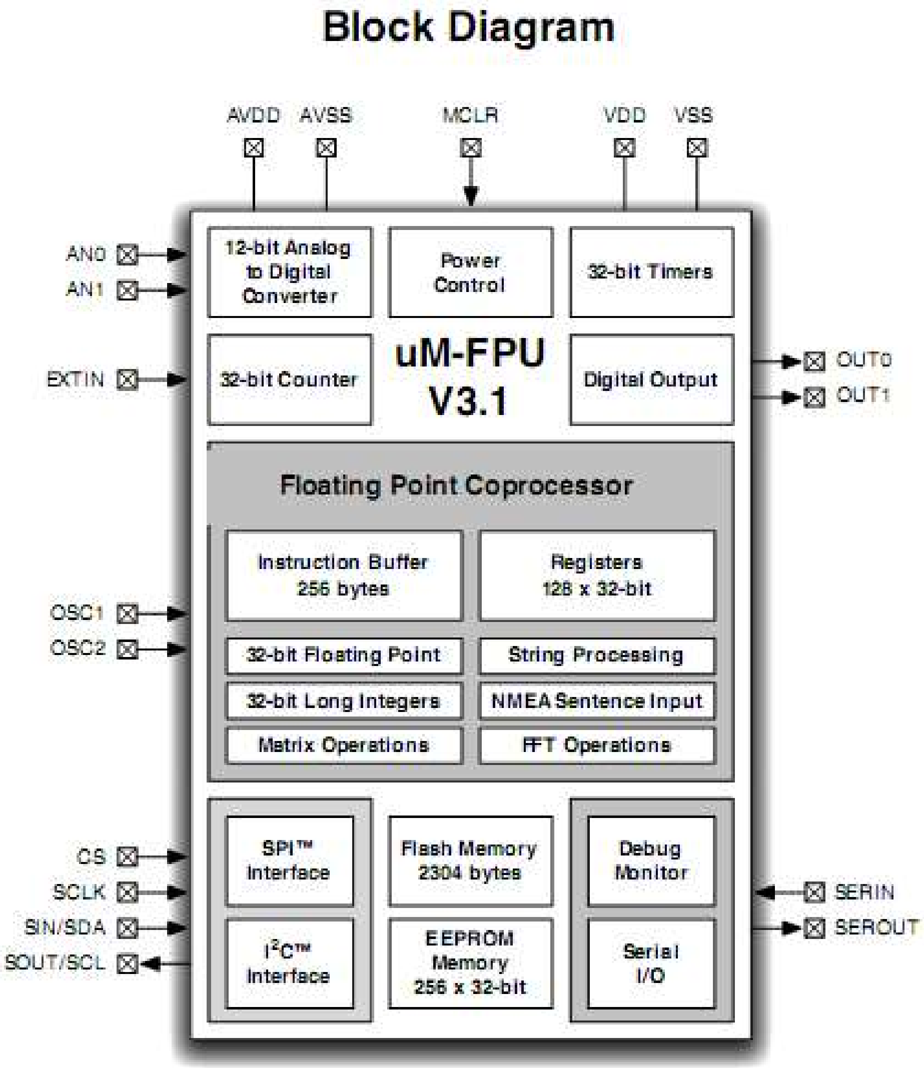

i387
Floating Point Unit
Status Word
Control Word
16
16
DBUS Interface
Digital Alignment and Operant Checking
Tag - Word
16
Internal Data Bus
32
D0 - D15
16
Data Buffer
32
Data FIFO
25 x 32 Bit
Registers
Exponent Adder
(16 Bit)
Operand B EXP
Register (16 Bit)
Operand A EXP
Register (16 Bit)
16
16
Exponent Busses
(0)
(1)
(2)
(3)
(4)
(5)
(6)
(7)
Register File (Stack)
8 x 80 Bit
Mantissa Busses
68
68
Operand Register
(A and B)
Mantissa Adder
(68 Bit) +
Sum Register
16 Bit Left/Right
Barrel Shifter +
68 Bit Register
Constant ROM
44 x 67 Bit
Instruction
Decoder
Micro
Instruction
Sequencer
Micro Instruction Bus
FPU Control +
CORDIX Nano-Machine
STATUS
ADDRESS &
BUS CONTROL
Bus
Control
Logic
Power
Management
Unit
CPUCLK2
NUMCLK2

80486DX2


Intel 80486DX2 Architecture
64 Bit Interunit Transfer Bus
32 Bit Data Bus
32 Bit Data Bus
Linear Address 32 Bit
Core Clock
Clock Multiplier
Clock
Barrel Shifter
Register File
ALU
32 Bit
Segmentation Unit
Descriptor Registers
Limit and Attribute PLA
Paging Unit
Translation Lookaside Buffer
PCD
PWT
2
20 Bit
Physical Address
Cache Unit
8 KByte Cache
32 Bit
32 Bit
32 Bit
Address Drivers
Write Buffers 4 x 32
Data Bus Transeivers
Bus Control
Request Sequenzer
Burst Bus Control
Bus Size Control
Cache Control
Parity Generation and Control
Boundary Scan Control
A31 - A2, BE3# - BE0#
D31 - D0
ADS#, W/R#, D/C#, M/IO#, PCD, PWT, RDY#, LOCK#, PLOCK#, BOFF#, A20M#, BREQ, HOLD, HLDA, RESET, SRESET, INTR, NMI, SMI#, SMIACT#, FERR#, IGNNE#, STPCLK#
BRDY#, BLAST#
BS16#, BS8#
KEN#, FLUSH#, AHOLD, EADS#
DP3 - DP0, PCHK#
TCK, TMS, TDI, TD0
128 Bit
Displacement Bus
32 Bit
Prefetcher
32 Byte Code Queue
(2 x 16 Byte)
Micro-Instruction
24 Bit
Code Stream
Instruction Decode
Decoded Instruction Path
Control & Protection Test Unit
Control ROM
FPU
Floating Point Register File

FPU

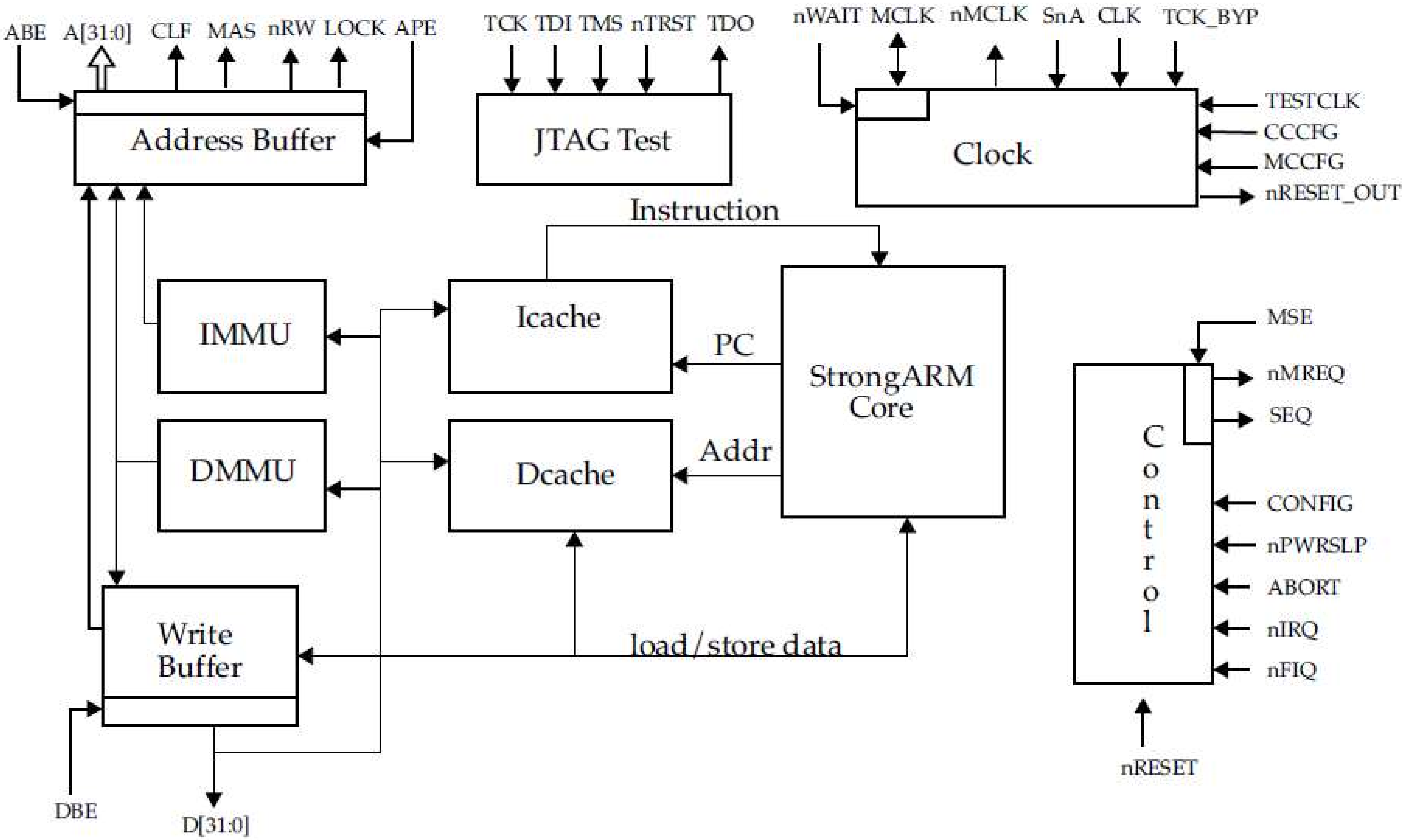

uMFPU
V3
www.HVWTech.com


Block Diagram
AVDD
AVSS
MCLR
VDD
VSS
AN0
AN1
EXTIN
OSC1
OSC2
CS
SCLK
SIN/SDA
SOUT/SCL
OUT0
OUT1
SERIN
SEROUT
12-bit Analog to Digital Converter
Power Control
32-bit Timers
32-bit Counter
uM-FPU V3.1
Digital Output
Floating Point Coprocessor
Instruction Buffer 256 bytes
Registers 128 x 32-bit
32-bit Floating Point
String Processing
32-bit Long Integers
NMEA Sentence Input
Matrix Operations
FFT Operations
SPI™ Interface
I²C™ Interface
Flash Memory 2304 bytes
EEPROM Memory 256 x 32-bit
Debug Monitor
Serial I/O

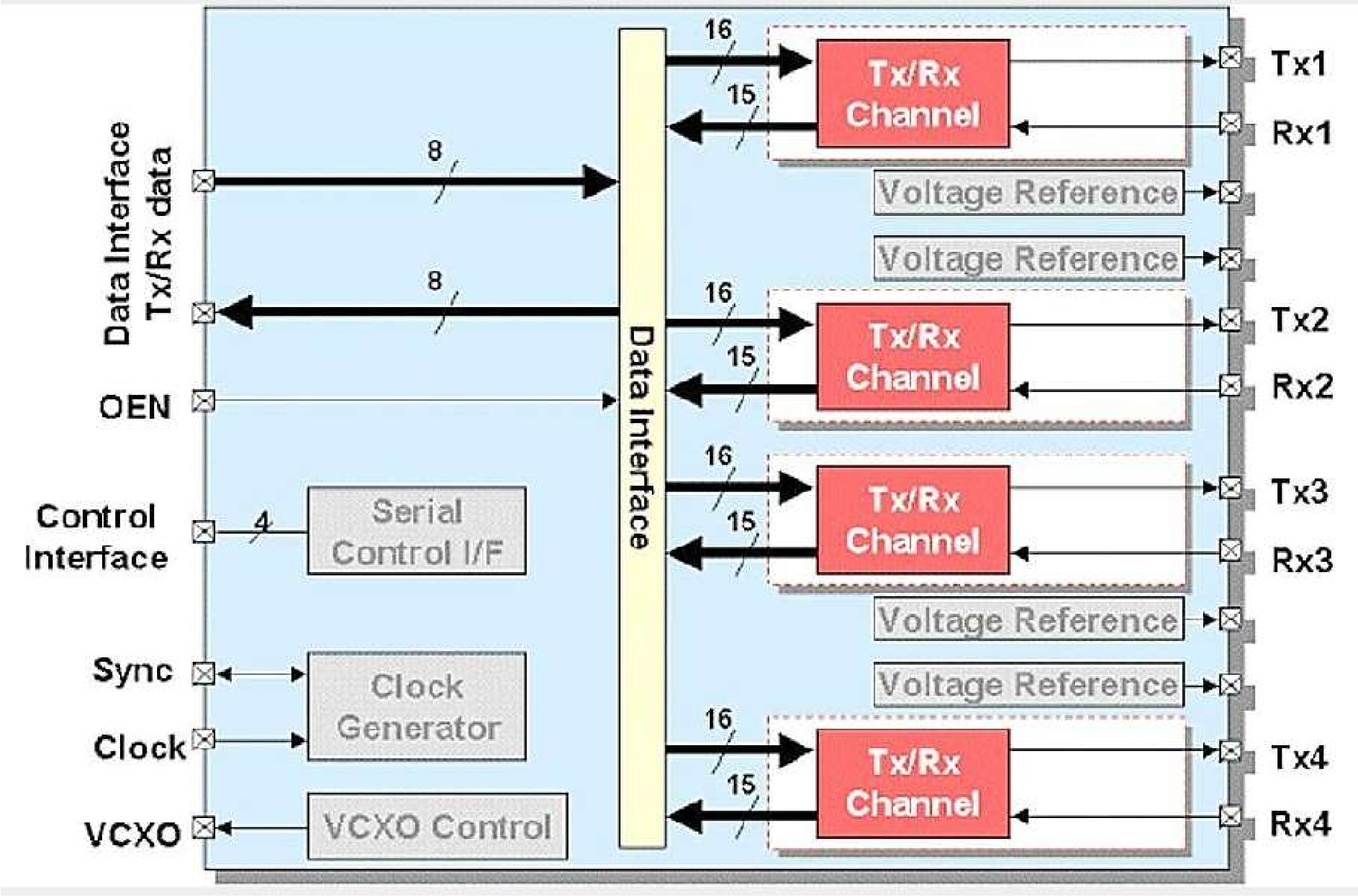
ARM Architecture

- The ARM is a 32-bit RISC instruction set architecture (ISA) developed by ARM Holdings. It was known as the Advanced RISC Machine, and before that as the Acorn RISC Machine.
- The ARM architecture is the most widely used 32-bit ISA in terms of numbers produced. They were originally conceived as a processor for desktop personal computers.
- The relative simplicity of ARM processors made them suitable for low power applications. This has made them dominant in the mobile and embedded electronics market as relatively low cost and small microprocessors and microcontrollers.
  - 2007: about 98% of the more than 1 billion mobile phones sold use at least one ARM processor.
  - 2009: ARM processors account for approximately 90% of all embedded 32-bit RISC processors.
- ARM processors are used extensively in consumer electronics, including PDAs, mobile phones, digital media and music players, hand-held game consoles, calculators and computer peripherals such as hard drives and routers.
- The ARM architecture is licensable.

Strong ARM


ABE
A[31:0]
CLF
MAS
nRW
LOCK
APE
Address Buffer
TCK
TDI
TMS
nTRST
TDO
JTAG Test
nWAIT
MCLK
nMCLK
SnA
CLK
TCK_BYP
Clock
TESTCLK
CCCFG
MCCFG
nRESET_OUT
Instruction
IMMU
Icache
PC
StrongARM Core
DMMU
Dcache
Addr
MSE
nMREQ
SEQ
Control
CONFIG
nPWRSLP
ABORT
nIRQ
nFIQ
nRESET
Write Buffer
load/store data
DBE
D[31:0]

Application Specific Standard Product (ASSP) Fujitsu



Performance Concepts

Oscillator(s), Power Saving Modes, Global Performance

PIC16F87XA - Oscillator Configurations

- The PIC16F87XA can be operated in four different oscillator modes.
- The user can program two configuration bits (FOSC1 and FOSC0) to select one of these four modes:
    1. **LP** Low-Power Crystal (32 kHz to 200 kHz)
    2. **XT** Crystal/Resonator (455 kHz to 4 MHz)
    3. **HS** High-Speed Crystal/Resonator (8 MHz to 20 MHz)
    4. **RC** Resistor/Capacitor (usually <= 8 MHz)

PIC16F87XA - Oscillator Configurations

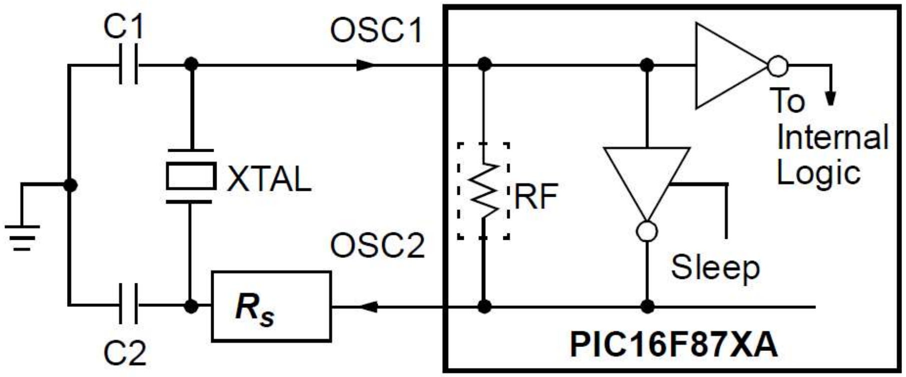


Figure 1: Crystal/ceramic resonator operation (HS, XT or LP OSC configuration)

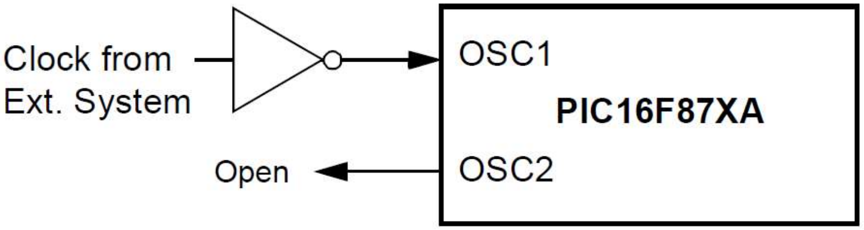
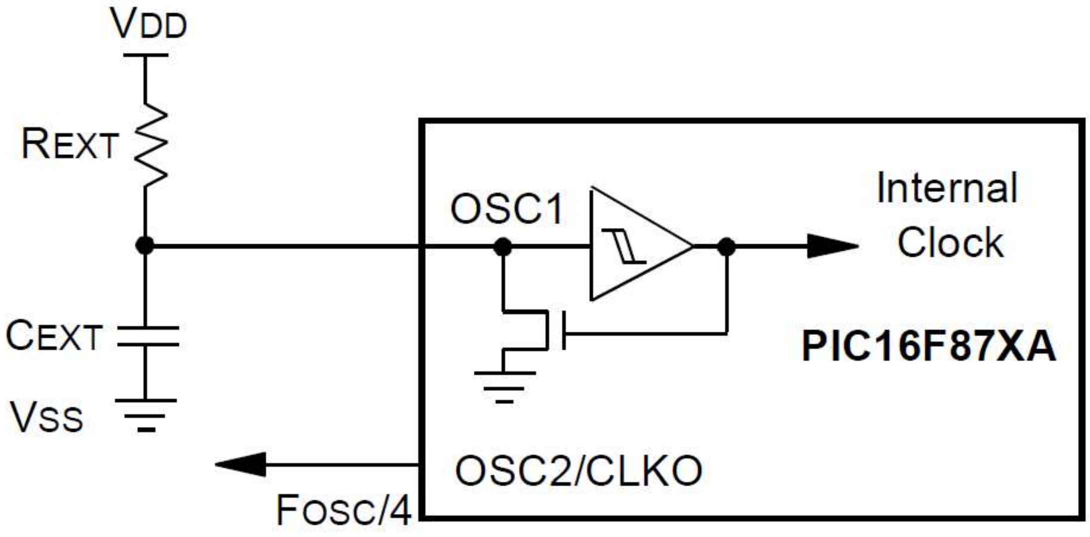


Figure 2: external clock input operation (HS, XT or LP OSC configuration)

PIC16F87XA - Oscillator Configurations



Recommended values: $3\ k\Omega \leq R_{EXT} \leq 100\ k\Omega$
$C_{EXT} > 20\ pF$

Figure 3: RC oscillator mode

PIC18F452 - Oscillator configurations

- The PIC18FXX2 can be operated in eight different Oscillator modes.
- The user can program three configuration bits (FOSC2, FOSC1, and FOSC0) to select one of these eight modes:
  1. **LP** Low Power Crystal
  2. **XT** Crystal/Resonator
  3. **HS** High Speed Crystal/Resonator
  4. **HS + PLL** High Speed Crystal/Resonator with PLL enabled
  5. **RC** External Resistor/Capacitor
  6. **RCIO** External Resistor/Capacitor with I/O pin enabled
  7. **EC** External Clock
  8. **ECIO** External Clock with I/O pin enabled

PIC18F452 - Oscillator configurations

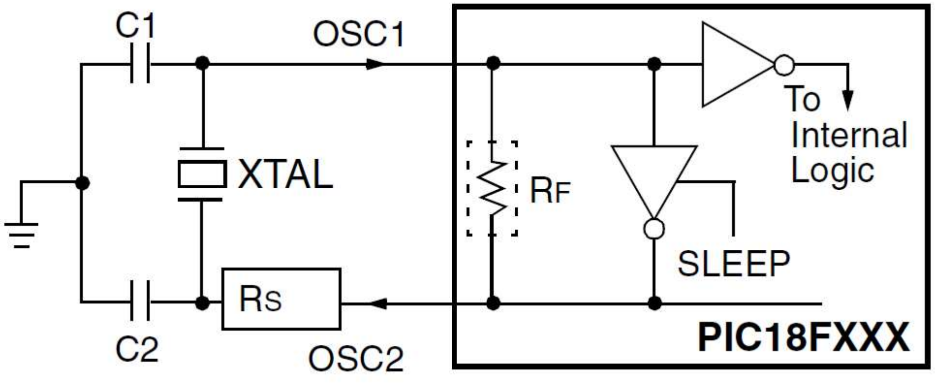


Figure 4: crystal/ceramic resonator operation (HS, XT or LP configuration)

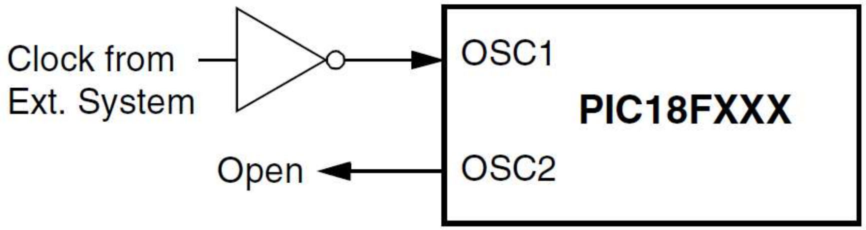


Figure 5: External clock input operation (HS, XT or LP OSC configuration)

PIC18F452 - Oscillator configurations

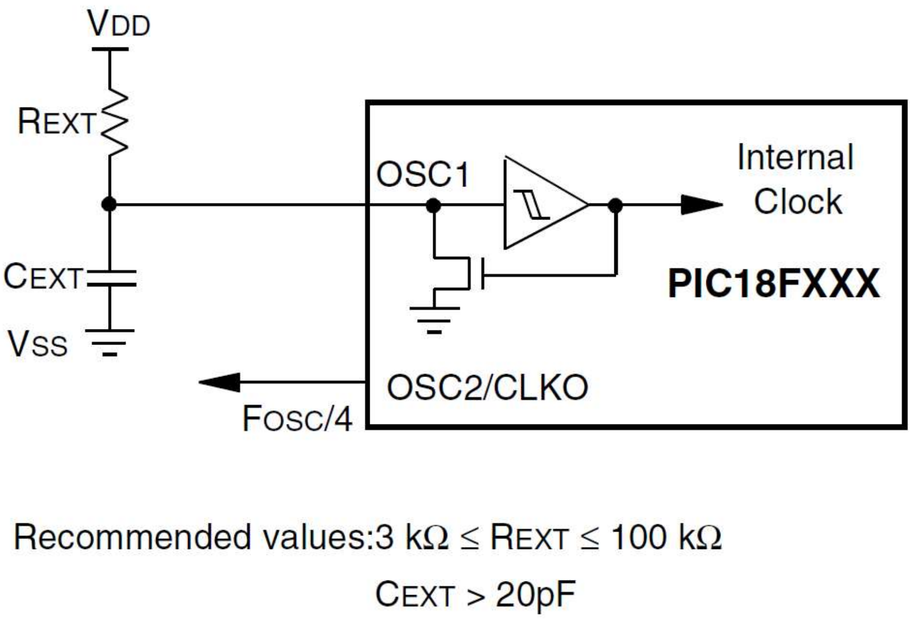
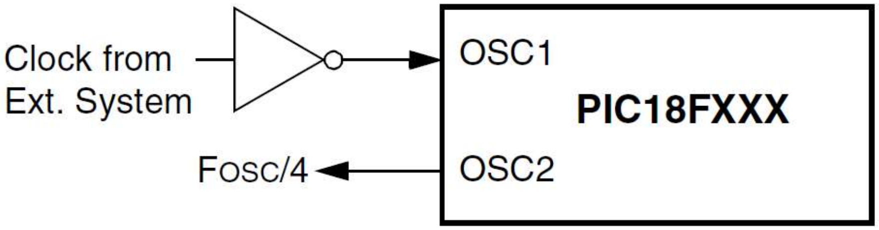


Figure 6: **RC** oscillator mode



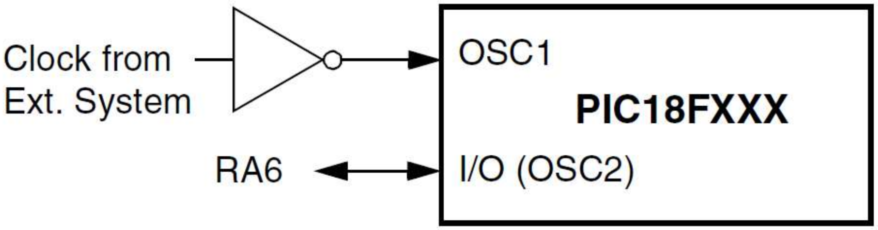


Figure 7: External clock input operation (**EC** configuration)

Figure 8: External clock input operation (**ECIO** configuration)

PIC18F452 - HS/PLL

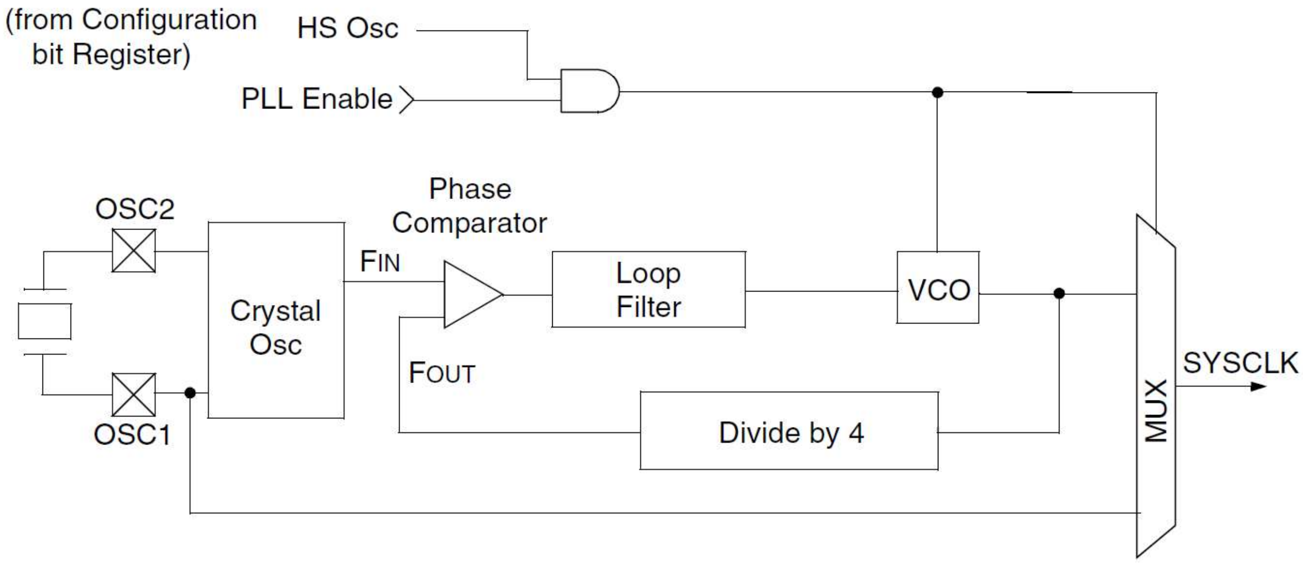


Figure 9: HS/PLL

Oscillator Switching Feature

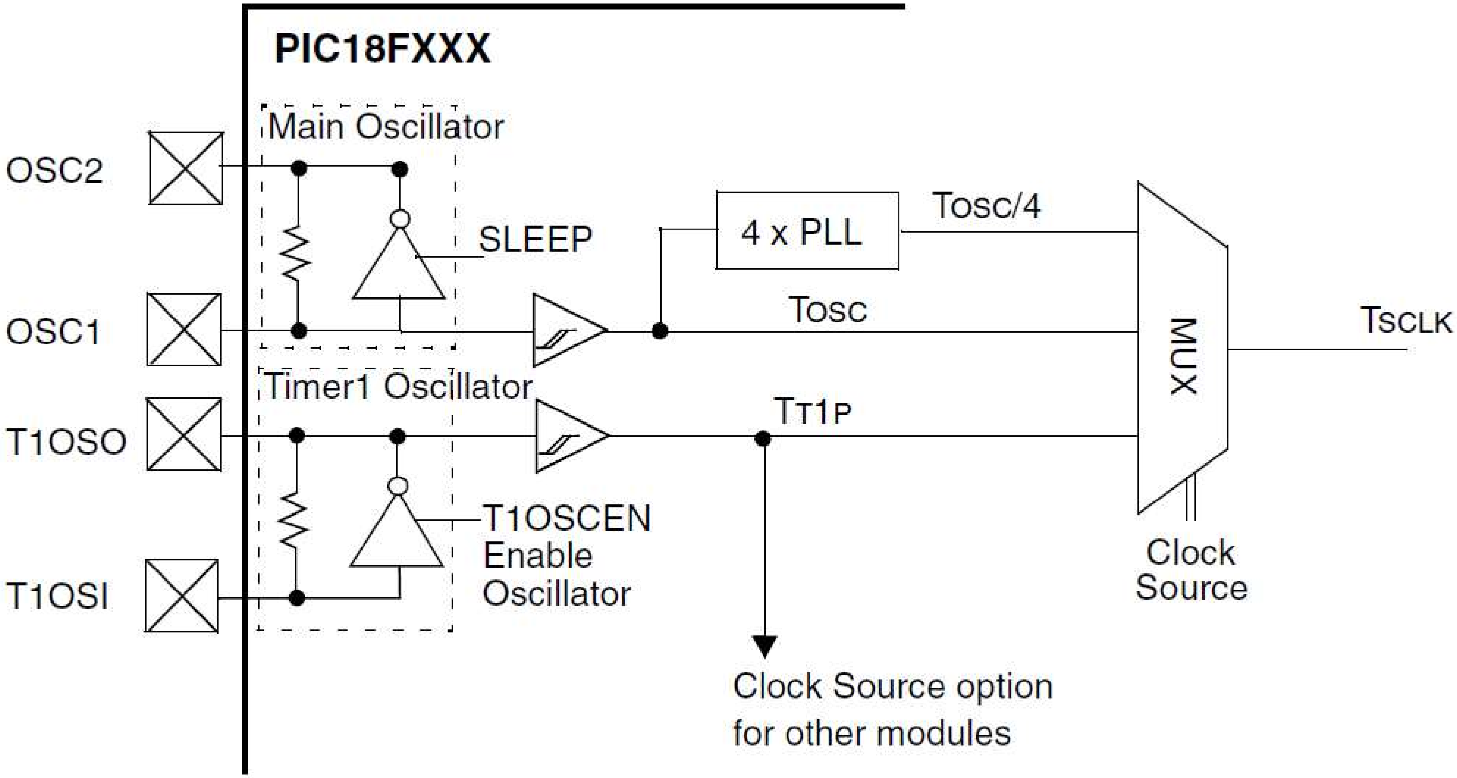


Figure 10: HS/PLL

Low Power

- Technology
- Lower *f*!
- Lower V_{DD}

$$P = \ P_S + P_D$$

$$P_S = n \cdot \ 100\mu W$$

$$P_D = f \cdot C_P \cdot {V_{DD}}^2$$

PIC18(L)F2X/4XK22

- Clock sources can be configured from external oscillators, quartz crystal resonators, ceramic resonators and Resistor-Capacitor (RC) circuits.
- the system clock source can be configured from one of three internal oscillators
  - with a choice of **speeds selectable via software**.
- Additional clock features include:
  - Selectable system clock source between external or internal sources **via software**.
  - Fail-Safe Clock Monitor (FSCM) designed to detect a failure of the external clock source (LP, XT, HS, EC or RC modes) and switch automatically to the internal oscillator.

PIC18(L)F2X/4XK22

- The primary clock module can be configured to provide one of six clock sources as the primary clock.

1. **RC** External Resistor/Capacitor
2. **LP** Low-Power Crystal
3. **XT** Crystal/Resonator
4. **INTOSC** Internal Oscillator
5. **HS** High-Speed Crystal/Resonator
6. **EC** External Clock

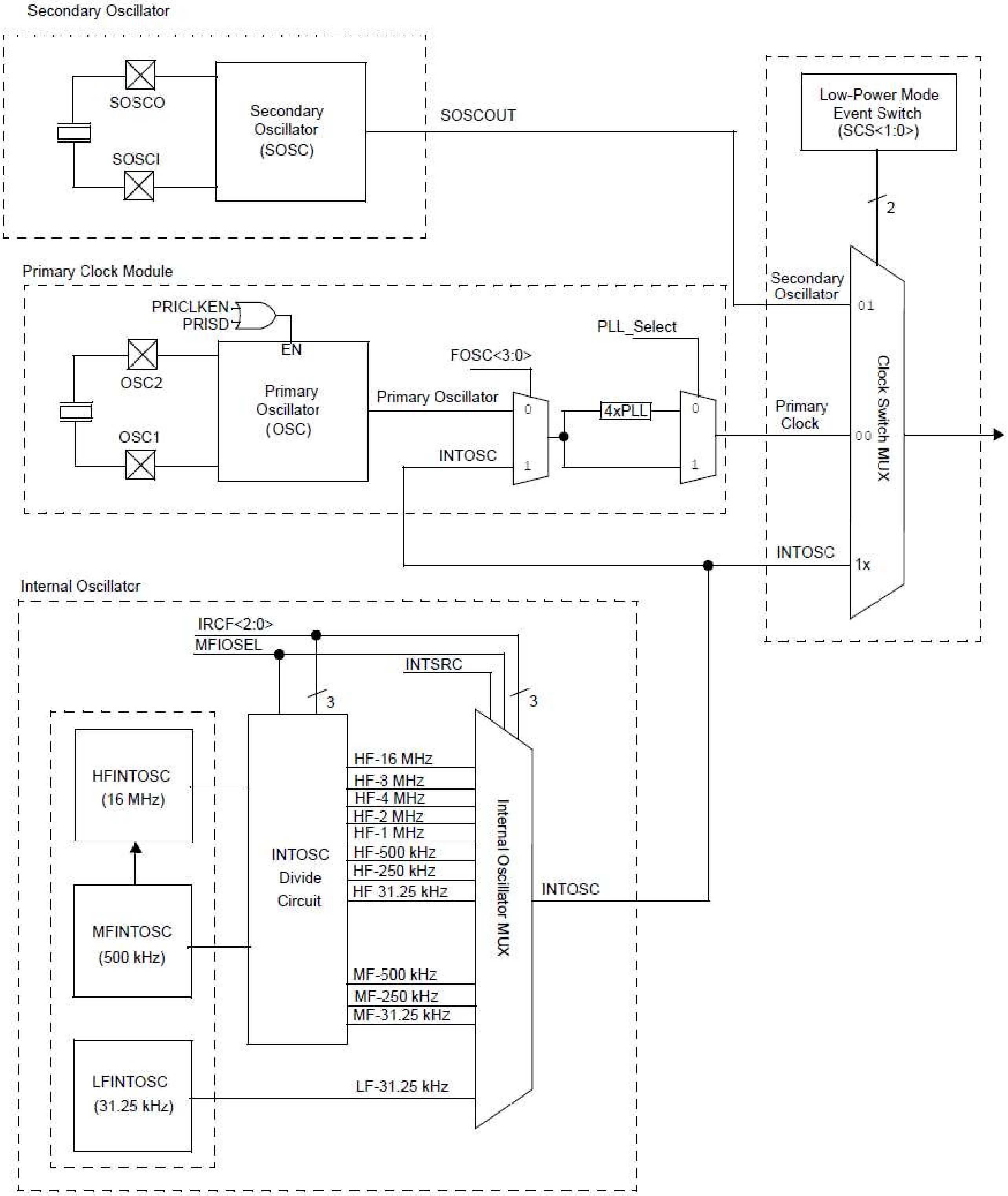


Figure 11: Block Diagram

PIC24FJ128GA010

- The oscillator system for PIC24FJ128GA010 family devices has the following features:
    - A total of four external and internal oscillator options as clock sources, providing 11 different clock modes
    - On-chip 4x PLL to boost internal operating frequency on select internal and external oscillator sources
    - Software-controllable switching between various clock sources
    - Software-controllable postscaler for selective clocking of CPU for system power savings
    - A Fail-Safe Clock Monitor (FSCM) that detects clock failure and permits safe application recovery or shutdown

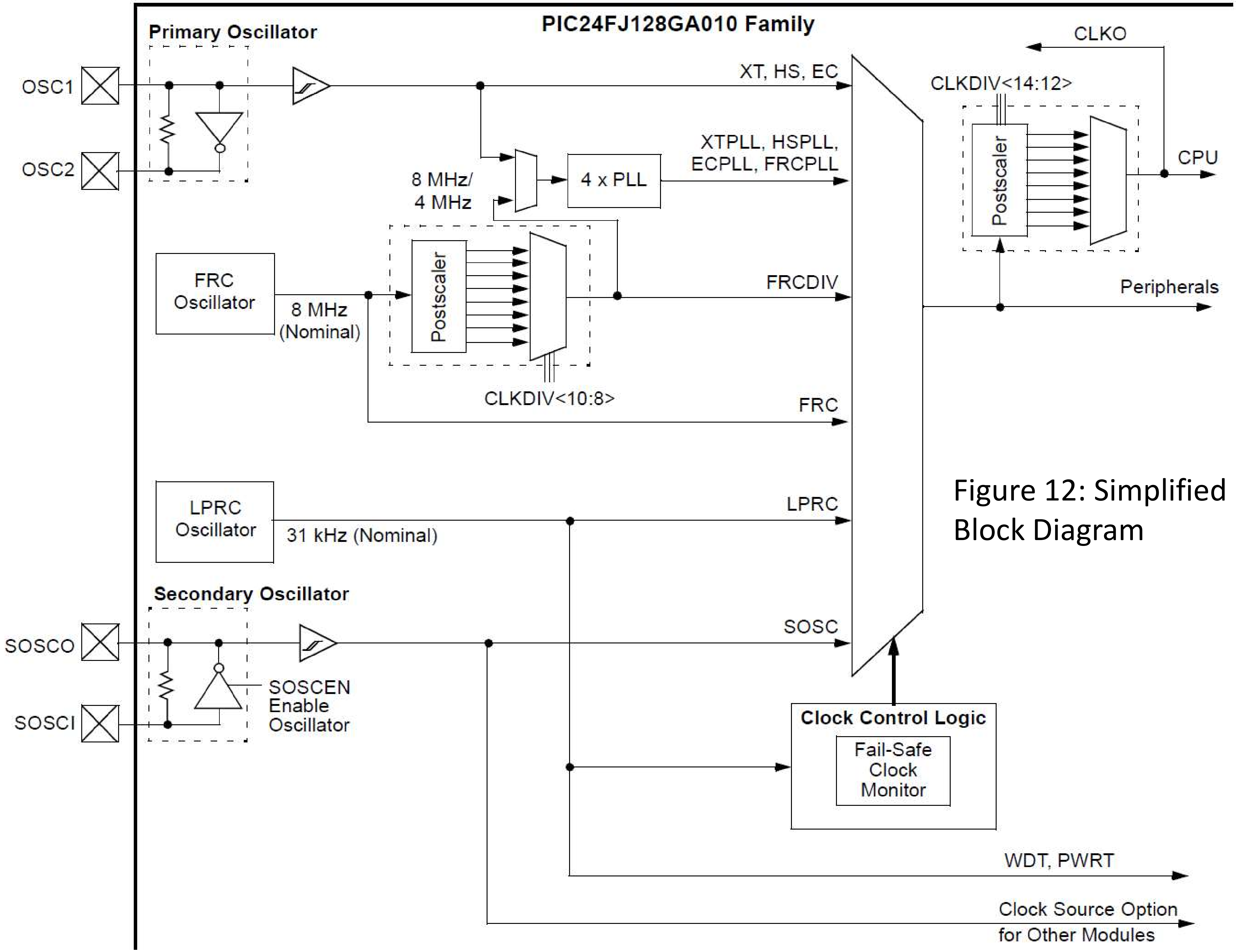


Figure 12: Simplified Block Diagram

PIC24FJ128GA010

- The system clock source can be provided by one of four sources:
  - Primary Oscillator (POSC) on the OSC1 and OSC2 pins
  - Secondary Oscillator (SOSC) on the SOSCI and SOSCO pins
  - Fast Internal RC (FRC) Oscillator
  - Low-Power Internal RC (LPRC) Oscillator

SLEEP MODE

- Sleep mode has these features:
  - The system clock source is shut down. If an on-chip oscillator is used, it is turned off.
  - The device current consumption will be reduced to a minimum provided that no I/O pin is sourcing current.
  - The Fail-Safe Clock Monitor does not operate during Sleep mode since the system clock source is disabled.
  - The LPRC clock will continue to run in Sleep mode if the WDT is enabled.
  - The WDT, if enabled, is automatically cleared prior to entering Sleep mode.
  - Some device features or peripherals may continue to operate in Sleep mode. This includes items, such as the input change notification on the I/O ports, or peripherals that use an external clock input. Any peripheral that requires the system clock source for its operation will be disabled in Sleep mode.
- The device will wake-up from Sleep mode on any of these events:
  - On any interrupt source that is individually enabled
  - On any form of device Reset
  - On a WDT time-out
- On wake-up from Sleep, the processor will restart with the same clock source that was active when Sleep mode was entered.

IDLE MODE

- Idle mode has these features:
  - The CPU will stop executing instructions.
  - The WDT is automatically cleared.
  - The system clock source remains active. By default, all peripheral modules continue to operate normally from the system clock source, but can also be selectively disabled.
  - If the WDT or FSCM is enabled, the LPRC will also remain active.
- The device will wake from Idle mode on any of these events:
  - Any interrupt that is individually enabled.
  - Any device Reset.
  - A WDT time-out.
- On wake-up from Idle, the clock is re-applied to the CPU and instruction execution begins immediately, starting with the instruction following the PWRSAV instruction or the first instruction in the ISR.

PIC32

- The PIC32 oscillator system has the following modules and features:
  - Four external and internal oscillator options as clock sources
  - On-chip Phase-Locked Loop (PLL) with a user-selectable input divider and multiplier, as well as an output divider, to boost operating frequency on select internal and external oscillator sources
  - On-chip user-selectable divisor postscaler on select oscillator sources
  - Software-controllable switching between various clock sources
  - A Fail-Safe Clock Monitor (FSCM) that detects clock failure and permits safe application recovery or shutdown

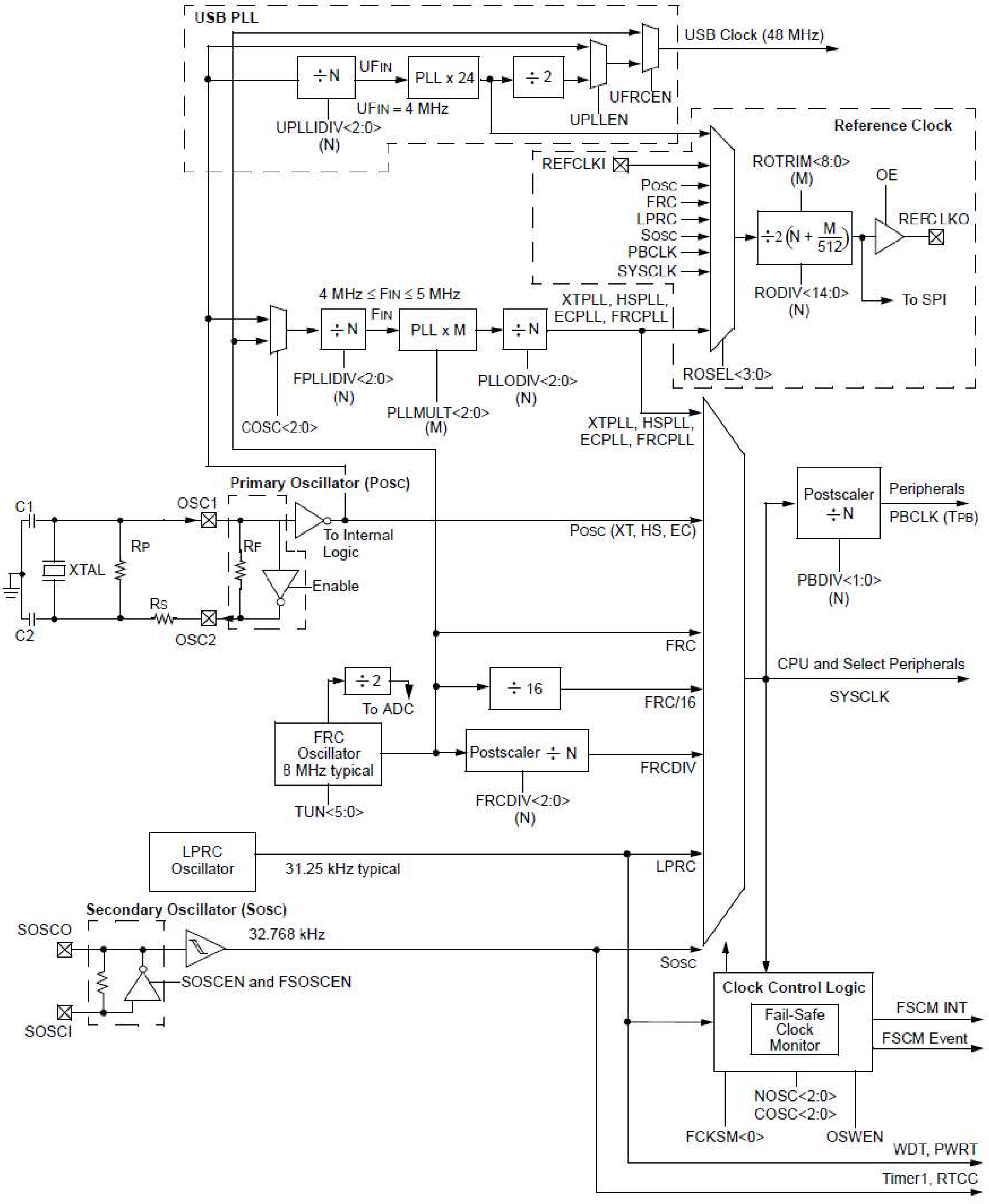


Figure 13: PIC32 Family Oscillator System Block Diagram

PIC32

- There are three main clocks in a PIC32 device:
  - System Clock (SYSCLK), used by the CPU and some peripherals
  - Peripheral Bus Clock (PBCLK), used by most peripherals
  - USB Clock (USBCLK), used by the USB peripheral
- The PIC32 clocks are derived from one of the following sources:
  - Primary Oscillator (POSC) on the OSC1 and OSC2 pins
  - Secondary Oscillator (SOSC) on the SOSCI and SOSCO pins
  - Internal Fast RC (FRC) Oscillator
  - Internal Low-Power RC (LPRC) Oscillator
- Each of the clock sources has unique configurable options, such as a PLL, an input divider and/or output divider.

PIC16 - PIC18 – PIC24 – PIC32

Parameter	PIC16	PIC18	PIC24	PIC32
T_{CY}	4 T_{OSC}	4 T_{OSC}	2 T_{OSC}	~1 T_{OSC}
$f_{OSC\ Max}$ (w PLL)	20 MHz	40 MHz	80 MHz	80 MHZ
(D)MIPS	5 MIPS	10 MIPS	40 MIPS	105 DMIPS