Microcomputer Concepts and the MCF54415 32-bit Microcontroller Unit

References:

- Freescale Semiconductor, Inc., “ColdFire Family Programmer’s Reference Manual”, Doc. No. CFRM, Rev. 3, July 2005.
- Freescale Semiconductor, Inc., “MCF5441x Reference Manual”, Doc. No. MCF54418RM, Rev. 4, Jan. 2012.

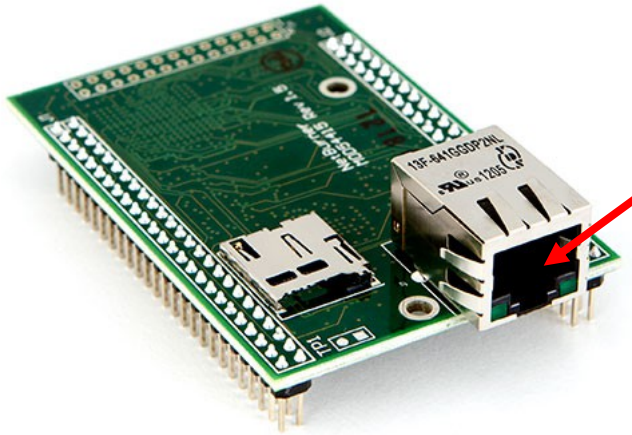
Figures and tables from the above documents have been included in these course notes with the permission of Freescale Semiconductor for educational purposes in ECE 315 only. The original Freescale Semiconductor documentation should be consulted to ensure accuracy.

In December 2015, NXP Semiconductors N.V. completed its acquisition of Freescale Semiconductor.

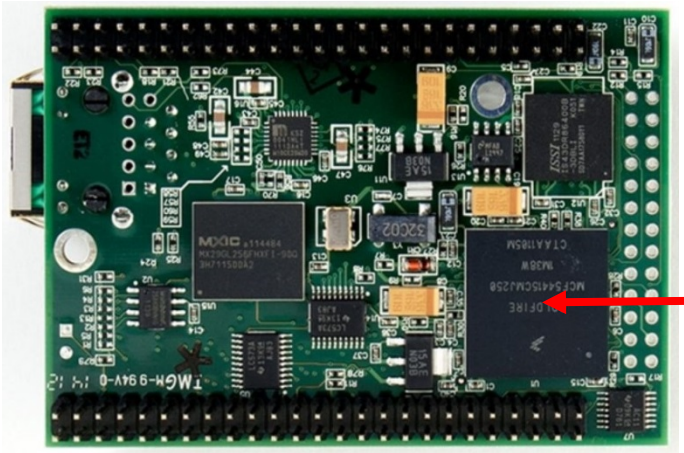
Microprocessor-related Terminology

- A **microprocessor** (μP) is a central processing unit (CPU) on one chip. The CPU is responsible for ***fetching***, ***decoding*** and ***executing*** binary machine language instructions that are stored in a memory.
- A **microcontroller unit** (MCU) is an *integrated circuit* (IC) that contains a microprocessor as well as other useful support circuits, such at timers, memory, input/output interfaces, direct memory access controllers, etc. The ECE 315 lab microcomputer is built using the Freescale MCF54415 MCU, which contains one Version 4 ColdFire 32-bit microprocessor.
- A **microcomputer** (μC) is a computer system that has been built around either a microprocessor or microcontroller unit chip. The ECE 315 lab microcomputer is the NetBurner MOD54415-100X.
- A **digital signal processor** (DSP) is a specialized microprocessor that has features (e.g., *fast multiply-accumulate instructions*, *parallel data signal paths*, etc.) that make it especially efficient at performing the the kinds of numerically-intensive calculations that are required in digital signal processing (e.g., in digital filters and image processing).

The ECE 315 Microcomputer: NetBurner MOD5441X



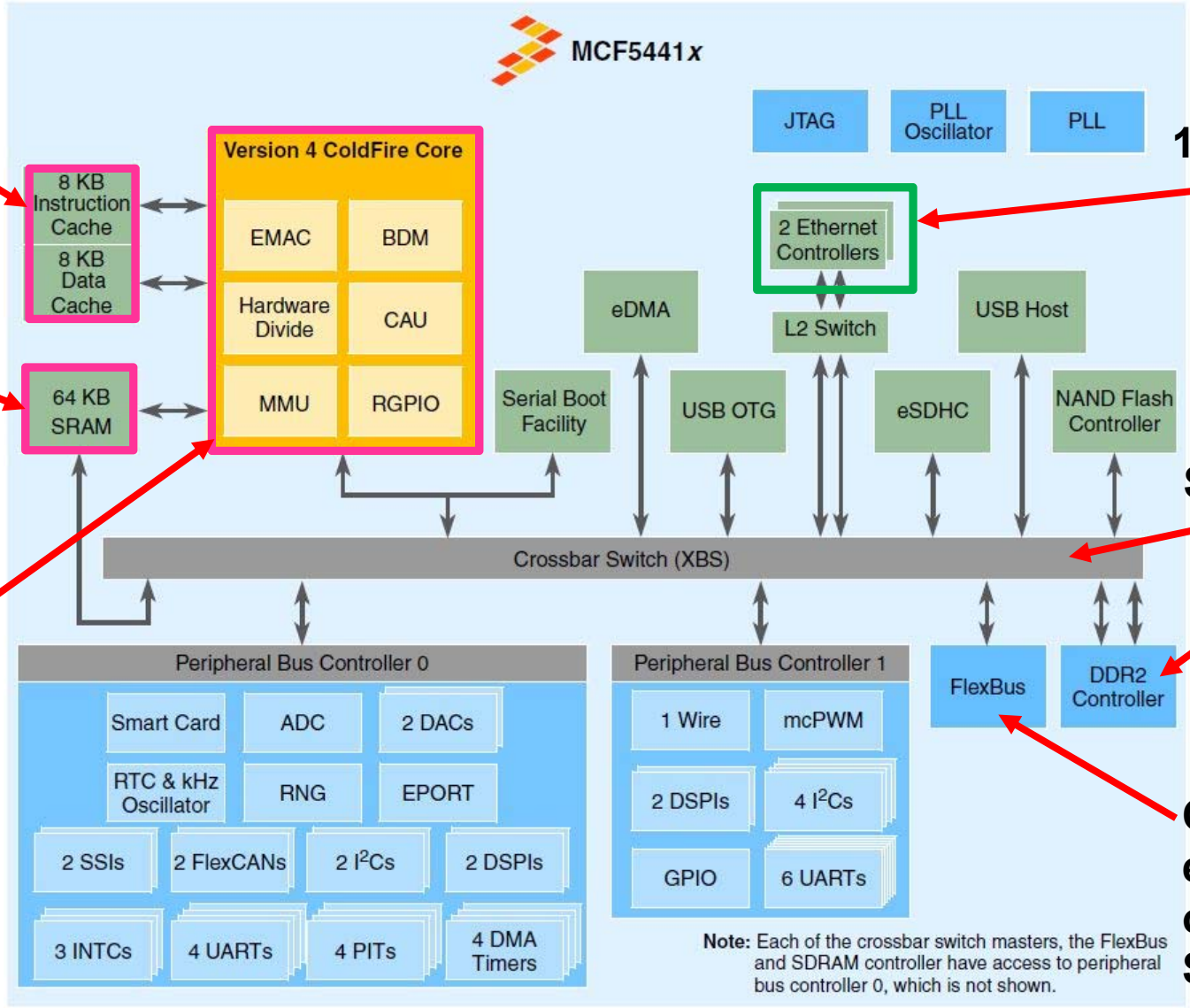
RJ45 jack
for Ethernet



NetBurner MOD54415-100IR µC
"Ethernet Core Module"

Freescale MCF54415 MCU
< 1100 mW power at 250 MHz

Architecture of the MCF5441x MCUs, x = 0,5,6,7,8



Split I$ and D$ caches

Fast single-cycle on-chip SRAM

V4 ColdFire 32-bit CPU / µP

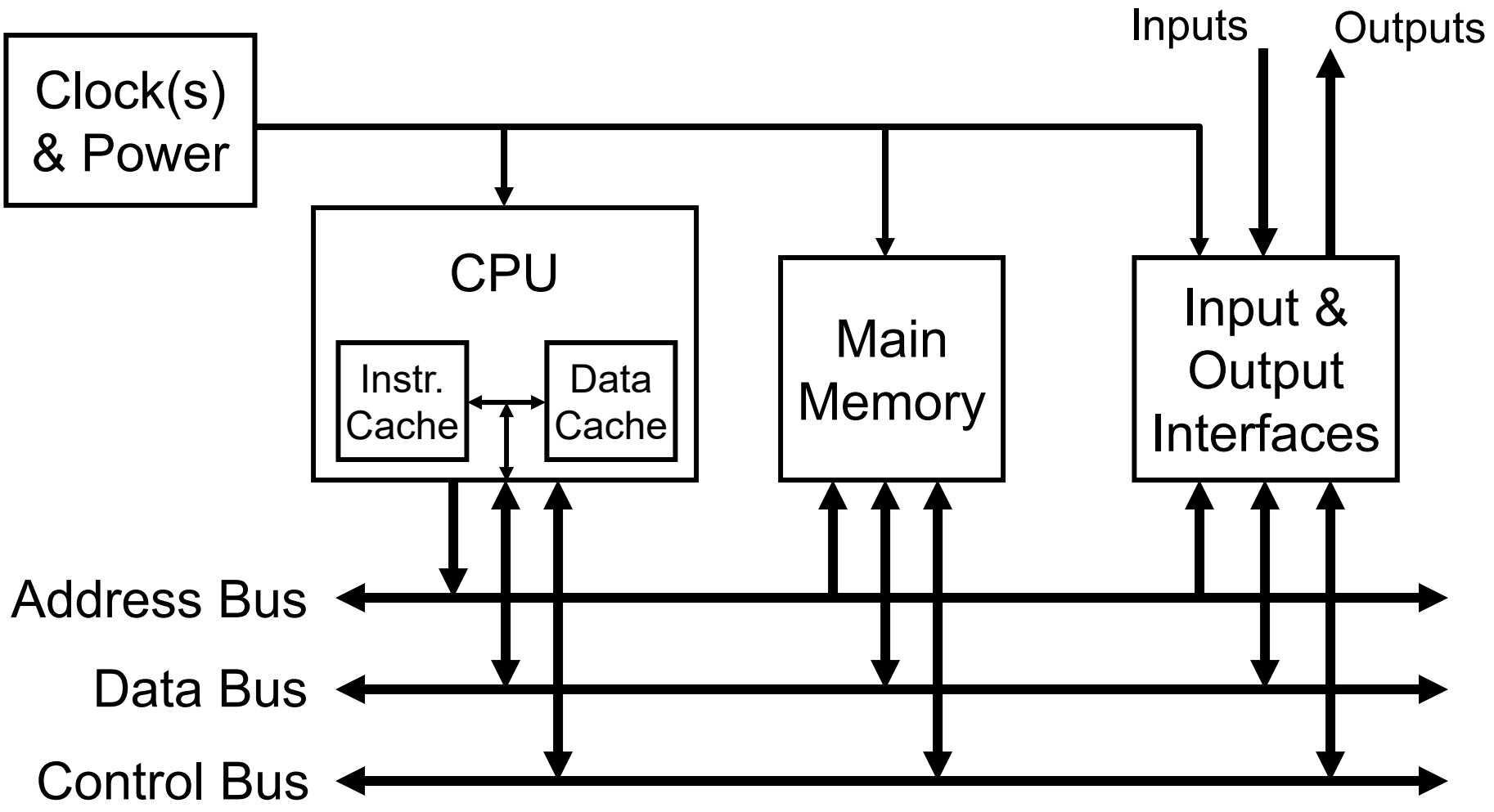
Two 10/100 Mbps Ethernet interfaces

On-chip System Bus

To/from system software

Off-chip extension of the System Bus

Microcomputer Architecture Using a Microprocessor with Separate Internal Instruction and Data Caches



The Instruction and Data Caches

- Cache memory is used in computers as a cost-effective way of *boosting performance* by speeding up memory accesses.
- Cache memory is a relatively *fast memory*, of relatively small capacity, that is *located physically close to the CPU*.
- The cache memory is used to store “local” copies of data and instructions that were retrieved previously from main memory as the software was executing on the CPU.
- Whenever the CPU fetches data or instructions, the cache is accessed to see if it contains a local, fast-access copy.
- The “*replacement rule*”, which is used to determine which data and instructions are evicted from (or retained in) the limited capacity cache, is tuned so that frequently used data and instructions are likely to be kept longer in the cache.

The System Bus

The system bus is usually composed of three sub-busses:

- The "*address bus*" is used by the CPU to tell the rest of the microcomputer system which address it is using for the present *read*, *write*, or *read-modify-write* bus operation.
- The "*data bus*" is used to communicate information between the different parts of the microcomputer.
  - During *reads*, data travels from the addressed location (in either the memory or the input/output devices) to the CPU.
  - During *writes*, data travels from the CPU to the addressed location (in either the memory or the input/output devices).
  - The data bus "width" (in bits) determines the data size of of the microcomputer system.
- The "*control bus*" contains various signals used to control and synchronize events in the microcomputer.

Microcomputer Memory

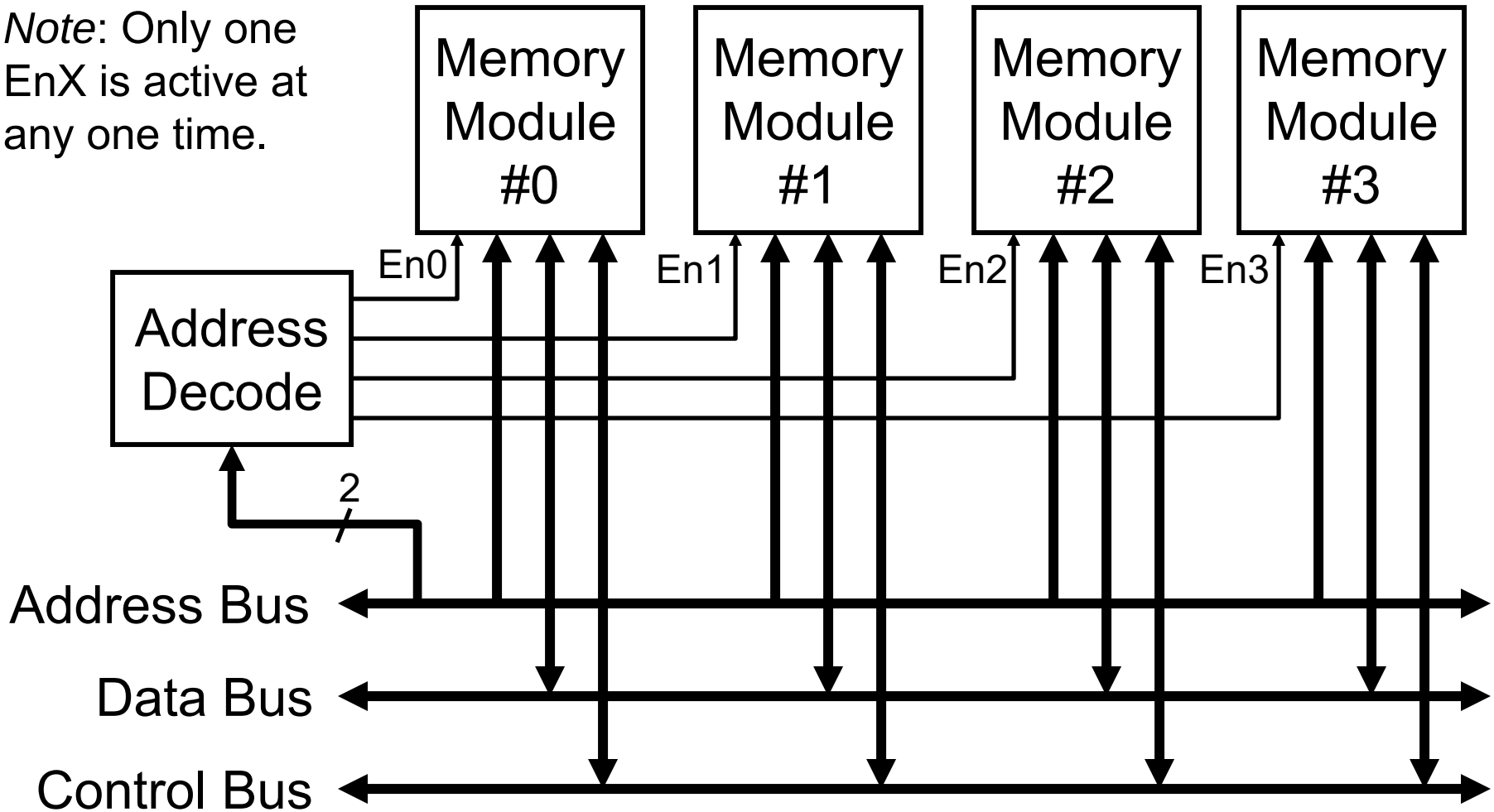
- A memory holds binary information, which consists of both *program instructions* and *data*. (In memory, all information is stored as 0's and/or 1's. Hexadecimal notation is often used to make binary values easier for humans to read and write.)
- Information "words" are identified in a memory using an address, which is just a binary (often written in hex) number. The "word size" of a computer is usually determined by the number of wires in the data bus (i.e., the data bus size).
- A *random-access memory* (RAM) allows the words stored in the RAM to be both *read* and *written*.
- A *read-only memory* (ROM) allows the words to be read, but not changed. The contents of a ROM are *fixed*. ROM is usually implemented today using flash memory.

Volatile vs. Nonvolatile Memory

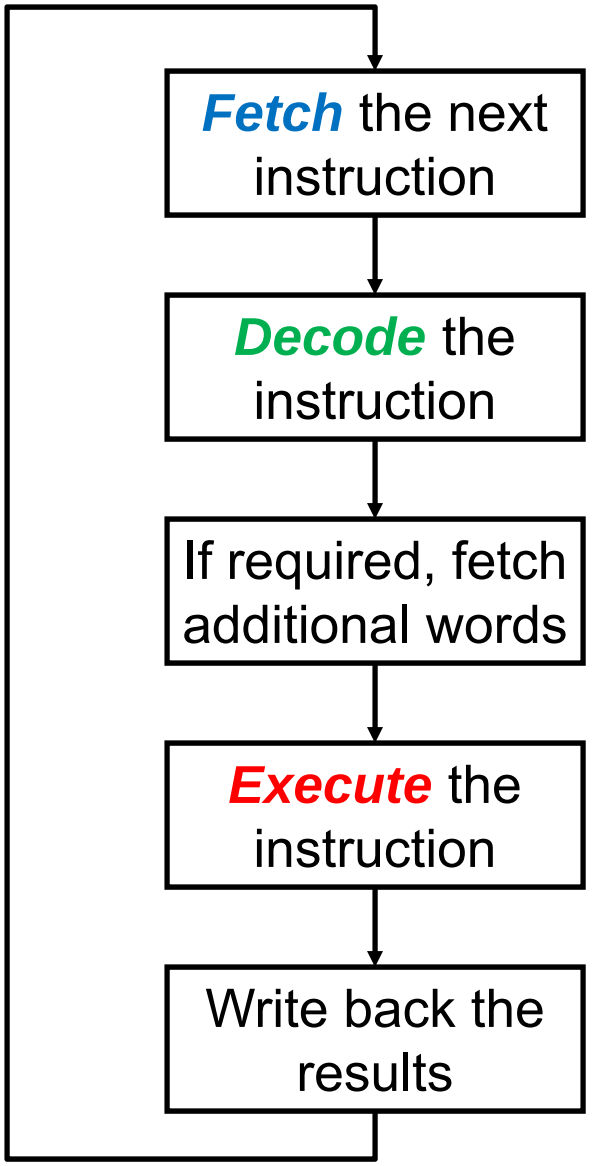
- Semiconductor memory is either *volatile* or *nonvolatile*.
- Volatile memory can retain stored data only as long as power is supplied.
  - *Static RAM* (SRAM) is fast but it is relatively power-hungry and expensive per bit. Smaller capacity than DRAM.
  - *Dynamic RAM* (DRAM) is cheap but relatively slow, and it requires refreshing to avoid losing stored data.
  - *Synchronous DRAM* (SDRAM) is a fast form of DRAM that uses a high-speed pipelined synchronous interface.
  - *Double data rate DRAM* (DDRx) is a fast form of SDRAM.
- *Nonvolatile memory* (NVM) can retain data even after power is lost. NVM is used to implement read-only memory (ROM).
  - *Flash memory* is currently the dominant NVM type.

A Simple Memory Subsystem

Note: Only one EnX is active at any one time.



The Instruction Fetch-Decode-Execute Cycle



- Place contents of PC on the address bus.
- Read first word of the present instruction off of the data bus (using a read bus cycle).

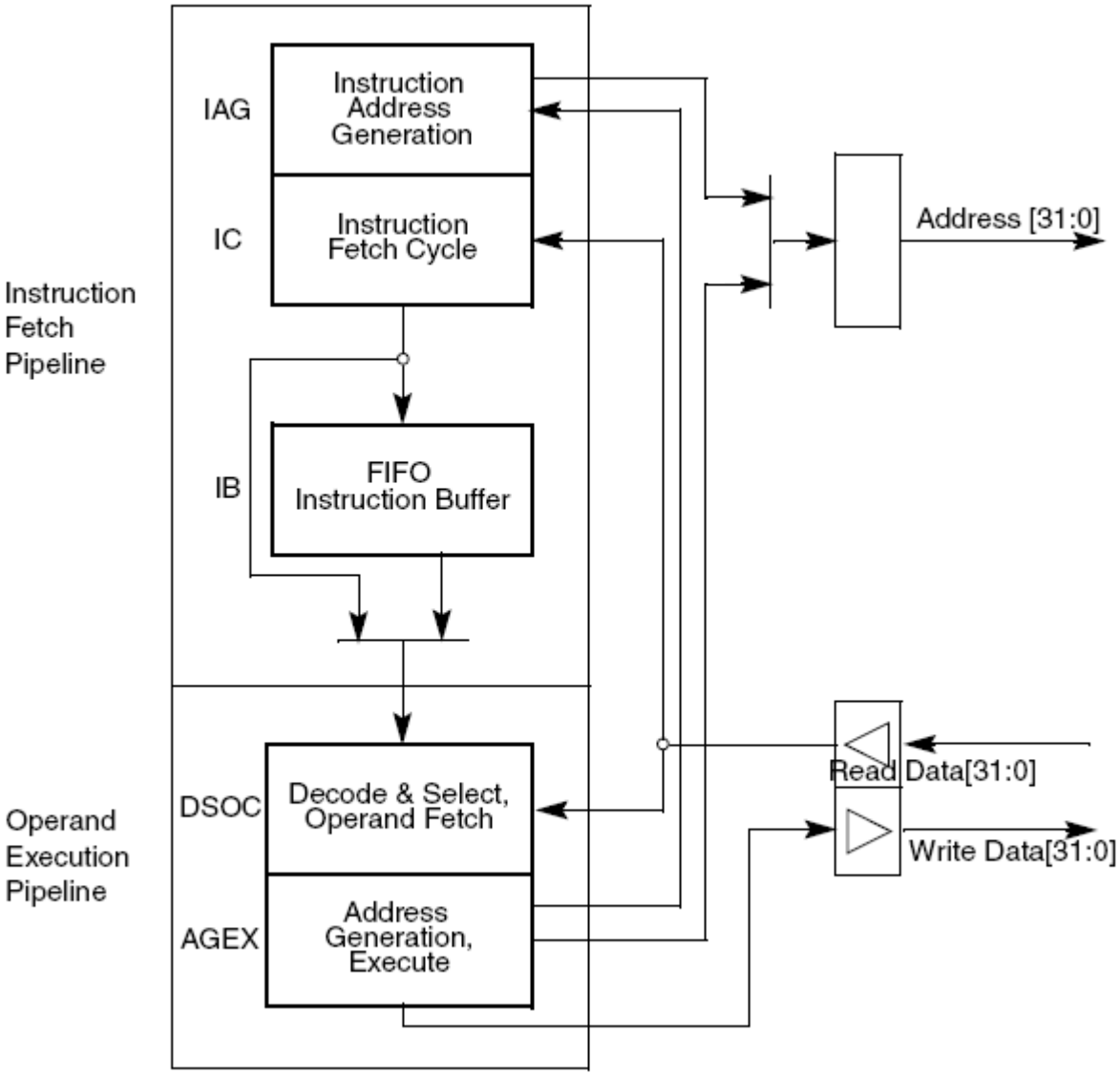
- Inspect the "operation code" in the first word of the instruction. Decide which kind of instruction is being performed.

- Update the PC, place PC contents on the address bus, and read additional data words (if necessary) using read bus cycles.

- Update all CPU registers (including the PC) according to the present register contents and the present instruction type.

- If necessary, update words in memory and I/O devices using write bus cycles.

Instruction Prefetching



Many modern CPUs, such as the Freescale MCF54415, have the ability to "prefetch" instructions from slow DRAM and to store them in a fast *first-in first-out* (FIFO) instruction buffer.

This is done to speed up the effective rate of instruction execution by reducing the average fetch time.

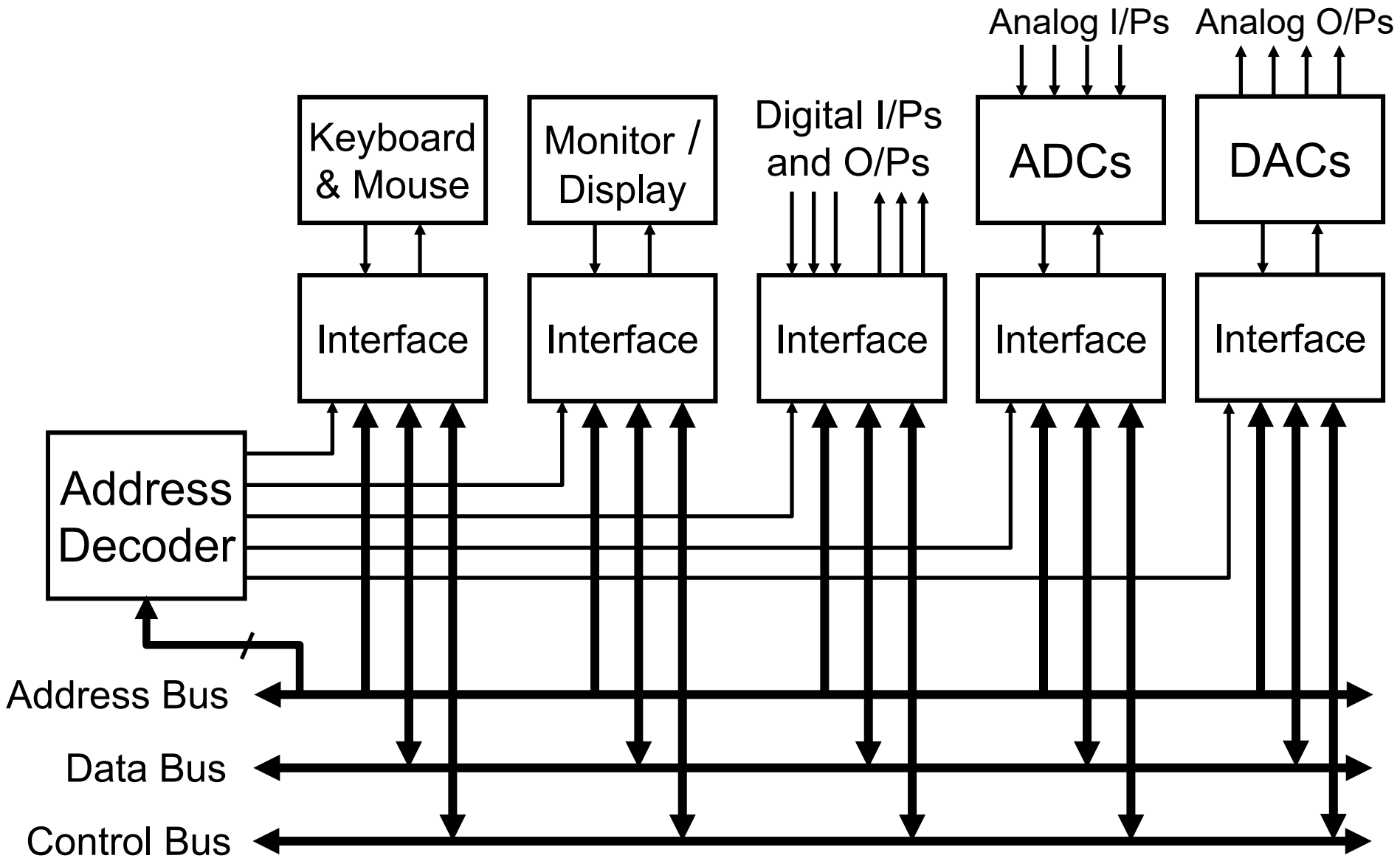
Input and Output Devices

- To be able to do useful work, a microcomputer needs to exchange information with its environment.

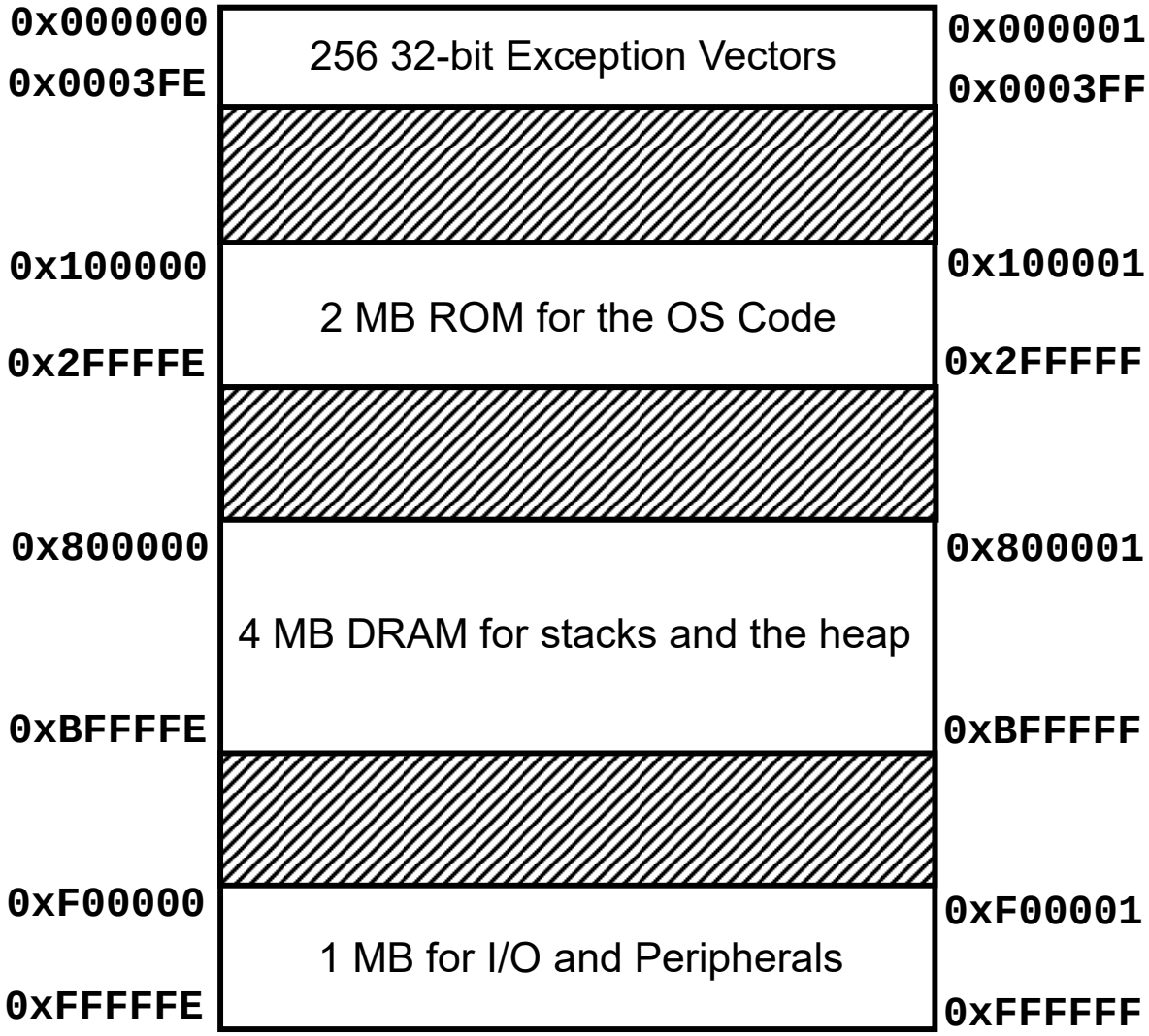
- ***Input devices*** allow information to be transferred from the environment into the microcomputer.
  - *Keyboard*, for inputting ASCII-encoded symbols.
  - *Mouse*, for inputting positions and command selections.
  - *Analog-to-digital converters* (ADCs), for converting measurements of analog signals into digital quantities.

- ***Output devices*** allow information to be transferred from the microcomputer out to the environment.
  - *Terminal Monitor or Screen*, for displaying graphical data and ASCII (or EBCDIC or Unicode) encoded symbols.
  - *Printers*, for producing hardcopy text and graphical output.
  - *Digital-to-analog converters* (DACs), for producing software-programmable analog signals.

Memory-Mapped I/O Interfaces



Ex: A Simple 68000-style16-MB Memory Map



A Selection of (now obsolete) Motorola Interface Chips

MC6800 Family (8-bit data, *synchronous* system bus):

- MC6821 Peripheral Interface Adaptor (PIA)
- MC6840 Programmable Timer Module (PTM)
- MC6843 Floppy Disk Controller (FDC)
- MC6845 Cathode Ray Tube Controller (CRTC)
- MC6850 Asynchronous Communications Interface Controller (ACIA)

M68000 Family (8/16/32-bit data, *semisynchronous* system bus):

- MC68120 Intelligent Peripheral Controller
- MC68175 VME Bus Controller
- MC68454 Intelligent Multiple Bus Controller
- MC68488 General Purpose Interface Adaptor
- MC68681 Dual Asynchronous Receiver/Transmitter (DUART)
- MC68590/802 Ethernet Controller chip set

Internal and External Interfaces

- When transistor budgets per IC were much smaller, it was necessary to implement most of the interface subsystems on additional ICs outside of the microprocessor IC.
- In fact, up until 1989/90, it was common to implement the floating-point instructions on a "*co-processor*" IC that was closely coupled with a separate microprocessor IC.
- As transistor budgets per economically manufacturable IC grew, more and more interface functions were brought onto the microprocessor IC, creating the microcontroller unit.
- Modern microcontroller units, such as the MCF54415, contain a large number of flexible *internal* (on-chip) interfaces. *External* (off-chip) interfaces can be accessed, if necessary, over an off-chip extension of the system bus.

CISC versus RISC CPUs

- The first 4-, 8- & 16-bit µPs were designed as *Complex Instruction Set Computers* (CISCs), with a relatively large number of instruction types and addressing modes to allow expert programmers to obtain compact assembly language code and the fastest possible execution.
- A more modern design philosophy is the *Reduced Instruction Set Computer* (RISC). Compared to a CISC, a RISC has a smaller number of instruction types and addressing modes. RISC designs thus require simpler hardware that can be more easily pipelined for higher clock rates than CISC designs.
- In practice, modern CPUs claim to be RISCs, but CISC features are commonly present in most modern CPUs to preserve compatibility with earlier CISC products.

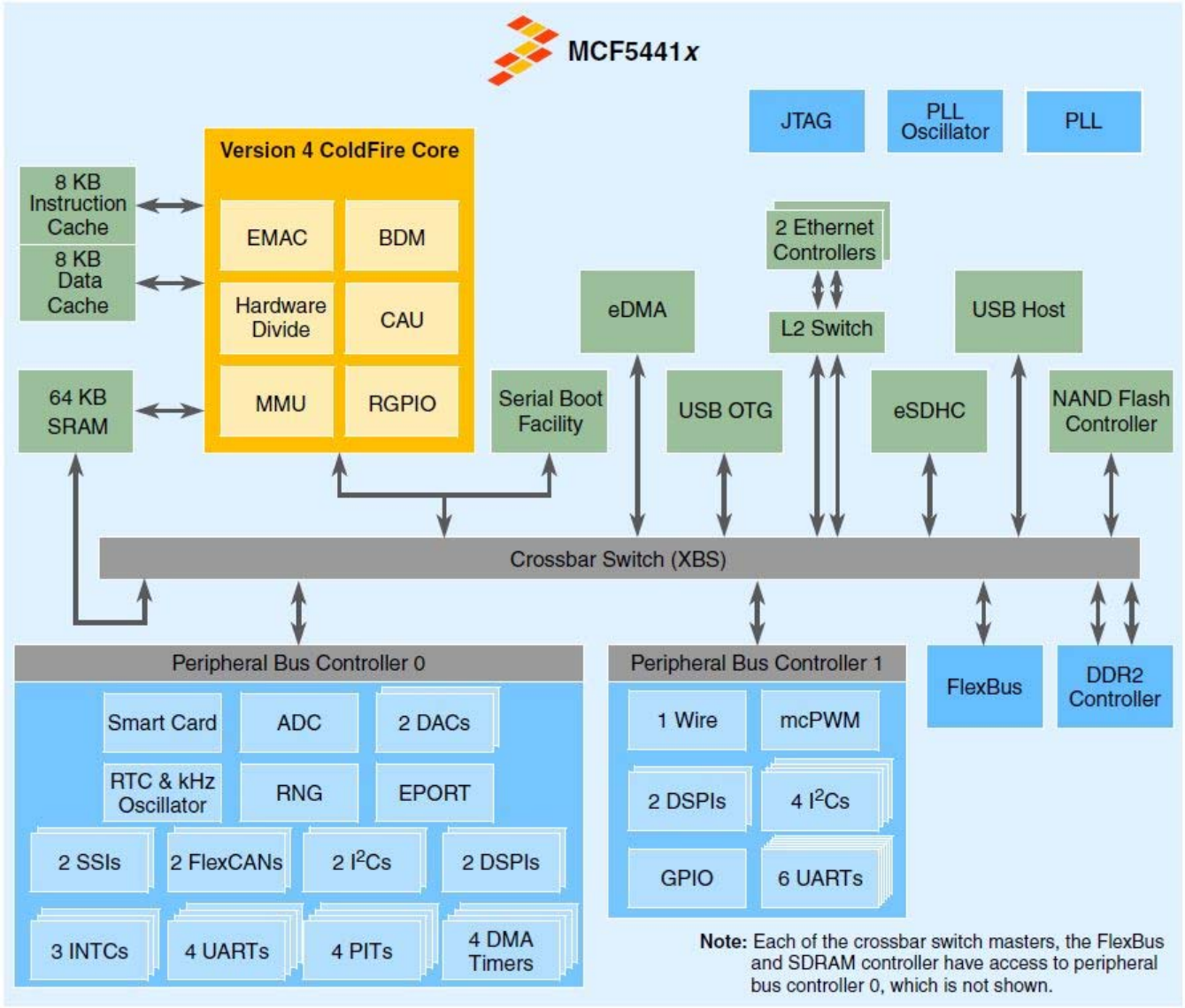
The ColdFire Family of 32-bit CPUs

- In 1994 Motorola introduced the Version 2 ColdFire μP core to update its 68xxx and CPU32 -class 16/32-bit μPs.
- The ColdFire is not object code compatible with 68xxx and CPU32 processors. However, object code from the older processors can be translated automatically into code that can be executed directly on ColdFire μPs.
- The ColdFire has fewer instructions (e.g., no binary-coded decimal instructions) and fewer addressing modes available for many instructions compared to the earlier 68xxx μPs.
- In 2004 the Semiconductor Products Sector of Motorola was spun off as Freescale Semiconductor, Inc. The ColdFire family continues to be developed by Freescale (now a part of NXP Semiconductor).

The ECE 315 Microcontroller Unit (MCU)

MCF54415 = 250-MHz V4 ColdFire µP + 16-Kbyte I/D cache +
64-Kbyte fast on-chip dual-ported SRAM +
64-channel DMA controller +
Synchronous DDR2 DRAM controller +
dual 10/100-Mbps Fast Ethernet Controllers (FEC) +
up to ten serial UARTs + USB 2.0 host interface +
CAN interfaces + I^2C interfaces + DSPI interfaces +
up to 87 general-purpose input/output pins +
watchdog timer + four 32-bit DMA timers +
four programmable periodic interrupt timers +
dual 4:1 muxed 12-bit ADCs and dual 12-bit DACs +
6 programmable chip selects + 5 IRQ inputs +
much more (see the MCF5441X Reference Manual)

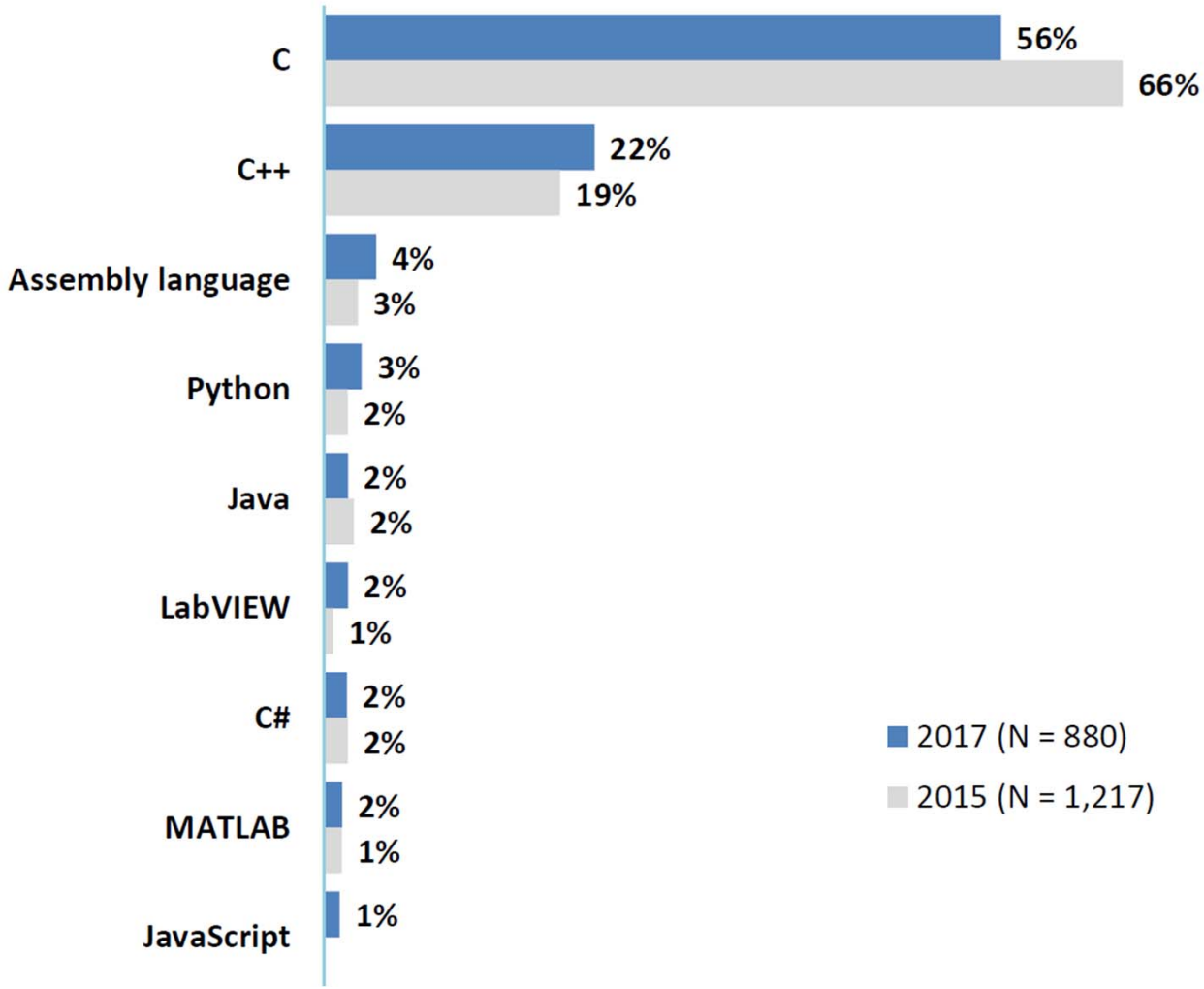
Architecture of the MCF5441x MCUs



Assembly Language vs. High-Level Language

- To maximize the effectiveness of the earliest generations of microprocessors, all programming was done in assembly language.
- As microprocessors became faster and software systems became larger, software was increasingly developed in high-level programming languages like C and C++.
- *Software designers are far more productive if they can use compiled high-level programming languages*.
- Software designed for the MCF5441x 32-bit micro-controller is likely to be entirely programmed in a high-level language like C. *Assembly language will only be used in the most time-critical portions of the operating system, and perhaps in some interrupt service routines*.

Languages Used to Implement Embedded Systems



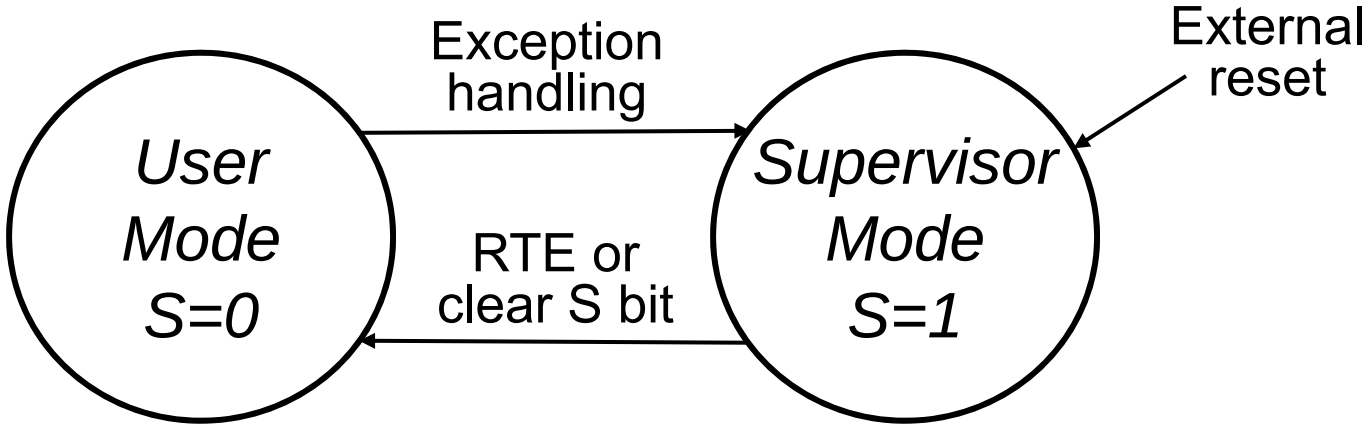
Courtesy of AspenCore's 2017 Embedded Markets Study

User and Supervisor Modes

- The 68xxx, CPU32 and ColdFire microprocessor families provide two different execution modes, called “User Mode” (S bit in the CPU Status Register is 0) and “Supervisor Mode” (S bit is 1).
- ***Supervisor Mode*** is intended for trusted operating system code and exception handling routines, while ***User Mode*** code is intended for ordinary software.
- Supervisor Mode uses a different stack pointer, can access a greater number of CPU registers, and can use some additional “privileged” instruction types.
- User Mode code requires little or no modification when moving to different processors within the family, but Supervisor Mode code typically needs some changes.

Transitions Between the User and Supervisor States

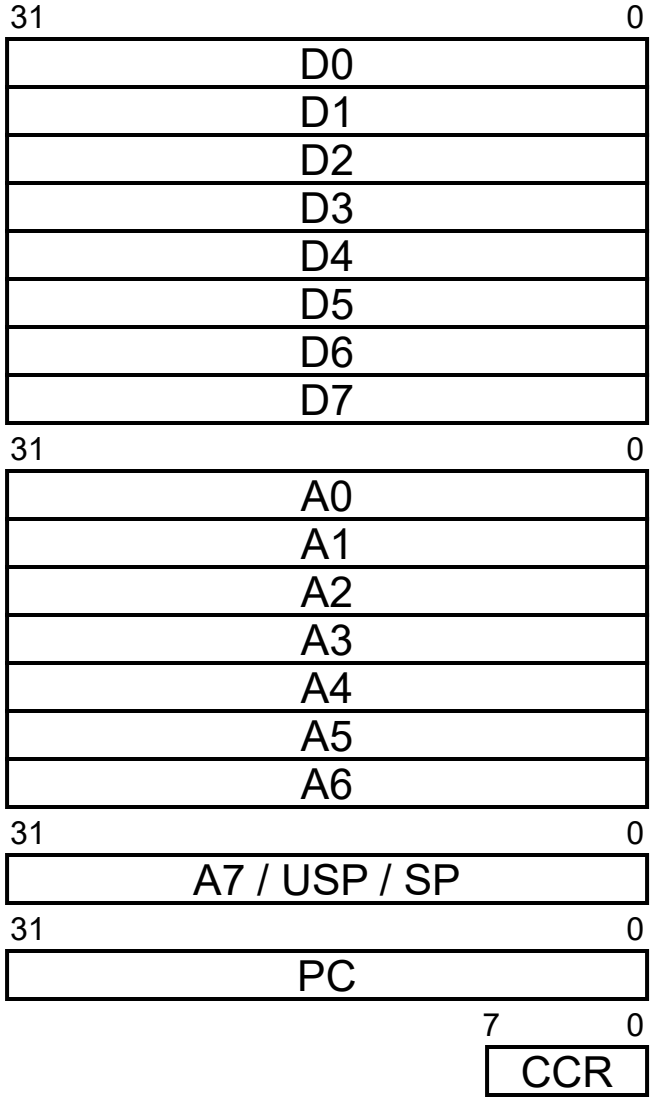
- When a CPU is reset by asserting the external hardware reset, it will start executing instructions in supervisor mode, with the S bit in the status register (SR) set to 1.
- All of the software could be executed in supervisor mode, but it is often desirable to restrict the privileges of at least some processes/tasks to user mode.



User Mode Programming Model (68xxx & ColdFire)

Can program the CPU as if it were a true 32-bit micro.

Interface logic handles transfers (1, 2 or 4 bytes) over the external 16-bit data bus.



Eight 32-bit Data Registers
- Dn.L => all 32 bits
- Dn.W => lower 16 bits
- Dn.B => lower 8 bits

Seven 32-bit Address Registers
- An.L => all 32 bits
- An.W => sign-extend the lower 16 bits

User Stack Pointer

Program Counter

Condition Code Register

Supervisor Mode Programming Model

- In supervisor mode, most of the registers are the same registers as in user mode: D0-D7, A0-A6, PC, CCR
- The stack pointer (A7 or SP) is a different register.
  The new register is the ***Supervisor Stack Pointer***, which can be identified using either A7, SP or SSP.
  The ***User Stack Pointer*** can still be accessed using USP.
- An extension of the CCR, the 16-bit ***Status Register*** (SR), can be accessed by supervisor mode programs.
- A 32-bit ***Vector Base Register*** (VBR) can be used to define a new base address for the Exception Vector Table.
- Eleven more 32-bit registers (CACR, ACR0-7, RAMBAR, MMUBAR) and 8-bit ASID control the CPU configuration.

CPU Registers for Supervisor Mode Only

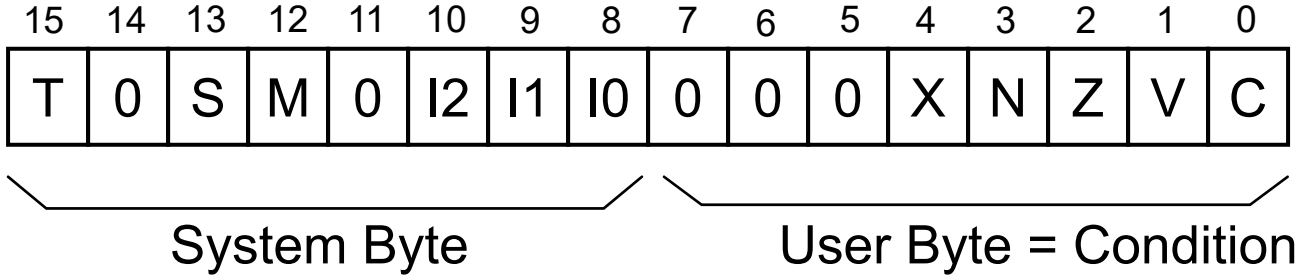
31:24	23:16	15:8	7:0	Mnemonic
—		Status Register		SR
Supervisor/User A7 Stack Pointer				A7
User/Supervisor A7 Stack Pointer				OTHER_A7
Vector Base Register				VBR
Cache Control Register				CACR
Access Control Register 0				ACR0
Access Control Register 1				ACR1
RAM Base Address Register				RAMBAR1

If S bit == 1, then A7 == SSP and OTHER_A7 == USP

If S bit == 0, then A7 == USP and OTHER_A7 == SSP

Note: OTHER_A7 is not visible to the software.

ColdFire Status Register



Bit 15: T => Trace mode enable/disable on every single instruction

Bit 13: S => Supervisor mode enable/disable

Bit 12: M => Master/interrupt state: cleared by interrupt exception, and can be set by the RTE or Move to SR instructions

Bits 10,9,8: I2,I1,I0 => Interrupt Priority Mask

Bit 4: X => Carry out bit for extended precision arithmetic

Bit 3: N => Negative result obtained

Bit 2: Z => Zero result obtained

Bit 1: V => Arithmetic overflow detected

Bit 0: C => Carry out bit (borrow out for subtraction)

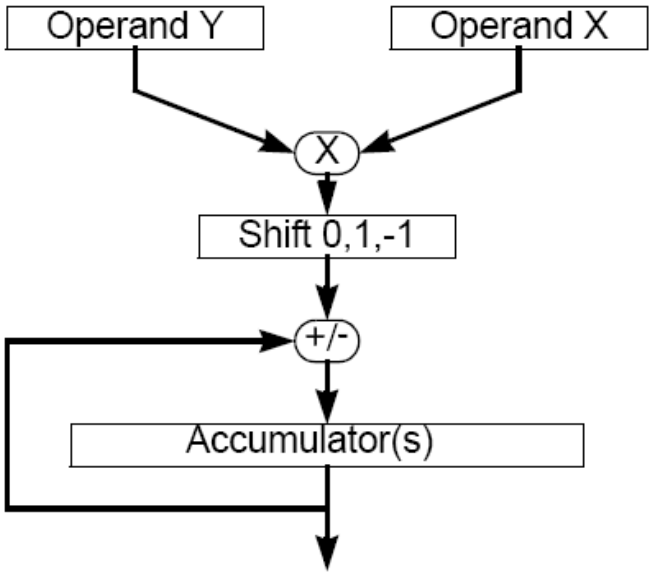
Common Multiply-Accumulate (MAC) Calculations

Four-tap Finite Impulse Response (FIR) Filter:

$$y(i) = \sum_{k=0}^{3} b(k)x(i-k) = b(0)x(i) + b(1)x(i-1) + b(2)x(i-2) + b(3)x(i-3)$$

N-tap Infinite Impulse Response (IIR) Filter:

$$y(i) = \sum_{k=1}^{N-1} a(k)y(i-k) + \sum_{k=0}^{N-1} b(k)x(i-k)$$



Enhanced Multiplier-Accumulate (EMAC) Unit

- To support customer requirements for high-speed MAC, some of the ColdFire processors have been provided with high-speed, pipelined MAC hardware that is used by the multiple instructions and new instructions that move data to and from the new MAC registers.
- The MCF54415 has an Enhanced Multiply-Accumulate (EMAC) unit that supports high-speed MAC operations optimized for 32-bit operands.
- The EMAC has a four-stage MAC pipeline, and provides four 48-bit accumulators (wider to support summations).
- New EMAC instructions are provided (see the Reference Manual for more details).

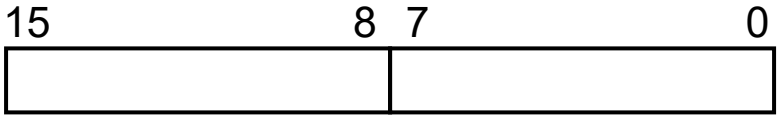
EMAC Registers

31:24	23:16	15:8	7:0	Mnemonic
MAC Status Register				MACSR
MAC Accumulator 0				ACC0
MAC Accumulator 1				ACC1
MAC Accumulator 2				ACC2
MAC Accumulator 3				ACC3
Extensions for ACC0 and ACC1				ACCext01
Extensions for ACC2 and ACC3				ACCext23
MAC Mask Register				MASK

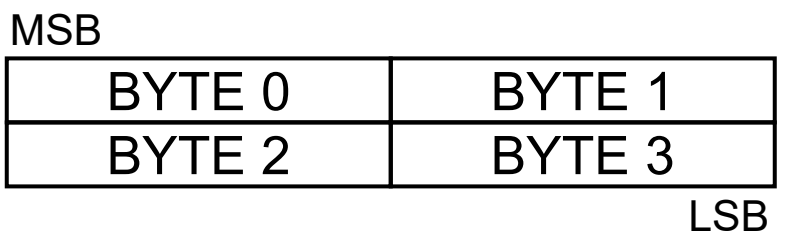
The MASK register is used to optionally constrain the address in MAC instructions to simplify accesses to data stored in circular queues (discussed later in the course).

Formats of the Native Data Types in Memory

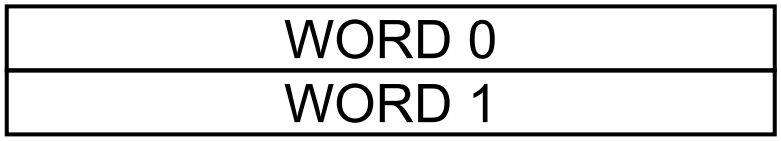
1) Bits



2) Bytes (8 bits)



3) Words (16 bits)



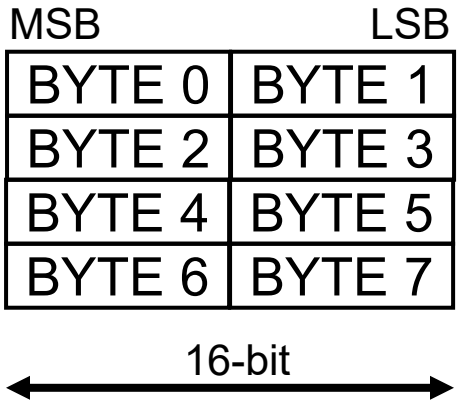
4) Longwords (32 bits)

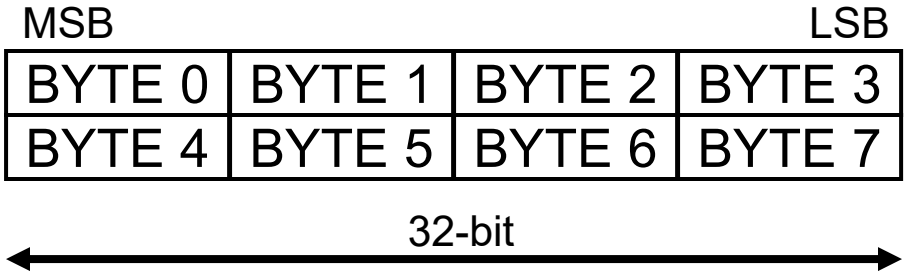
Note: The Binary-Coded Decimal (BCD) native data type that was supported in the M68000 and CPU32 is not present in the ColdFire.

Aside: Byte Address Order Conventions

Big Endian

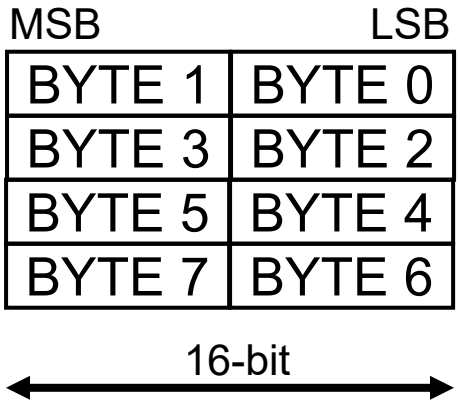
(IBM, Motorola/Freescale, HP, Sun)

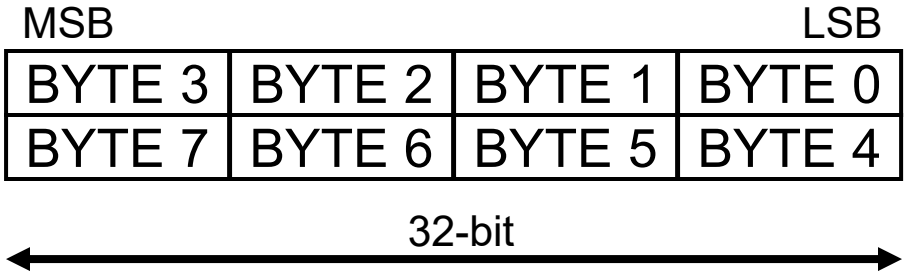




Little Endian

(Intel, AMD, early ARM CPUs)





Format of Machine Language Instructions

- ColdFire machine language instructions are variable in length: they can occupy *one*, *two* or *three* whole adjacent words, depending on the instruction type and the operands.
- *Note*: The original CISC M68000 has instructions that contained from one to seven whole words. The ColdFire instructions have less variability in length. This simplifies the implementation of the CPU.
- As in the M68000, ColdFire instructions are "word-aligned" in memory: i.e., they must start on even addresses.
- The first word in an instruction is the *operation word*. The left-justified *operation code* ("opcode") in the operand word is from 4 to 16 bits long.

ColdFire Opcode Map

Bits 15–12	Hex	Operation
0000	0	Bit Manipulation/Immediate
0001	1	Move Byte
0010	2	Move Longword
0011	3	Move Word
0100	4	Miscellaneous
0101	5	ADDQ/SUBQ/Scc/TPF
0110	6	Bcc/BSR/BRA
0111	7	MOVEQ/MVS/MVZ
1000	8	OR/DIV
1001	9	SUB/SUBX
1010	A	MAC/EMAC instructions/MOV3Q
1011	B	CMP/EOR
1100	C	AND/MUL
1101	D	ADD/ADDX
1110	E	Shift
1111	F	Floating-Point/Debug/Cache Instructions

Addressing Modes

- The source and destination operands of machine language instructions can be specified in many different ways called “addressing modes”.
- ***Simple addressing modes*** are available for specifying operands in terms of data values, register contents, or the contents of specified memory locations.
- ***Compound addressing modes*** are used to specify the operands in terms of two or more components. This is convenient for accessing common data structures.
- Some addressing modes are implied by the instruction type, and do not need to be encoded separately.

ColdFire Addressing Modes

Addressing Modes	Syntax	Mode Field	Reg. Field	Data	Memory	Control	Alterable
Register Direct							
Data	Dn	000	reg. no.	X	—	—	X
Address	An	001	reg. no.	—	—	—	X
Register Indirect							
Address	(An)	010	reg. no.	X	X	X	X
Address with Postincrement	(An)+	011	reg. no.	X	X	—	X
Address with Predecrement	–(An)	100	reg. no.	X	X	—	X
Address with Displacement	(d_{16},An)	101	reg. no.	X	X	X	X
Address Register Indirect with Scaled Index and 8-Bit Displacement	(d_8,An,Xi*SF)	110	reg. no.	X	X	X	X
Program Counter Indirect with Displacement	(d_{16},PC)	111	010	X	X	X	—
Program Counter Indirect with Scaled Index and 8-Bit Displacement	(d_8,PC,Xi*SF)	111	011	X	X	X	—
Absolute Data Addressing							
Short	(xxx).W	111	000	X	X	X	—
Long	(xxx).L	111	001	X	X	X	—
Immediate	#<xxx>	111	100	X	X	—	—

Simple Addressing Modes

1) **Immediate Data** #<value>

   Depending on the operand size specified by the instruction the data is obtained from:

   .B => the low order byte in the operation word
   .W => the next word following the operation word
   .L => the next longword following the operation word

2) **Quick Immediate Data** #<imm3> or #<imm8>

   Similar to immediate data, except that the value of the operand is restricted to either 3 bits or 8 bits, depending on which quick instruction is used. Instruction execution is a bit faster because the operand is embedded in the operation word.

Simple Addressing Modes (cont'd)

3) **Absolute Short Address** (0xNNNN).W
The included 16-bit integer is sign-extended to 32-bits to produce the address of the operand in memory.

4) **Absolute Long Address** (0xNNNNNNNN).L
The two included words are concatenated to form the 32-bit address of the operand in memory.

5) **Data Register Direct** Dn
The operand is contained in the specified data register.

6) **Address Register Direct** An
The operand is contained in the specified address register.

Indirect Addressing Modes

7) **Address Register Indirect** (An)
Operand is stored in memory at the address contained in the given address register An.

8) **Address Register Indirect with Postincrement** (An)+
Same as address register indirect, except that the contents of An are incremented at the end of instruction execution by 1, 2 or 4 if the operand size is byte, word or long word, respectively.

9) **Address Register Indirect with Predecrement** -(An)
Same as address register indirect, except that before any operands are selected by the instruction, the contents of An are decremented by 1, 2 or 4 if the operand size is byte, word or long word, respectively.

Compound Addressing Modes

10) Address Register Indirect with Displacement

(<d16>,An)

Similar to address register indirect, except that a 16-bit 2's-complement displacement <d16> is added to the address retrieved from the given address register An to give the address of the operand in memory.

11) Address Register Indirect with Scaled Index and 8-bit Displacement

(<d8>,An.s,Xn.s*scale) where scale = 1, 2, 4 or 8

Similar to address register indirect, except that an 8-bit 2's-complement displacement <d8> and the (possibly scaled) contents of an index register (Xn.s = An.W, An.L, Dn.W or Dn.L) are added to the address from the given register An to give the address of the operand in memory.

Compound Address Modes (cont'd)

12) **Program Counter with Displacement** (<d16>,PC)
The given 16-bit 2's-complement displacement <d16> is sign-extended to 32 bits and then added to the contents of the program counter to obtain the memory address of the operand.

13) **Program Counter with Scaled Index and 8-bit Displacement**
(<d8>,PC,Xn.s*scale) where scale = 1, 2, 4, or 8
Similar to program counter with displacement, except that the displacement <d8> is 8-bit and the contents of an index register (Xn.s = An.L, An.W, Dn.L or Dn.W) are added to the program counter to obtain the memory address of the operand. The .W index register values are sign-extended to 32 bits before being added in.

14) Implied Register Addressing Mode

- Some instructions imply that certain CPU registers contain the source and/or the destination operand(s).
  JSR, BSR, RTS, RTE, etc.
- In such instructions, there is no need to encode the operands explicitly as the addressing mode.
- Motorola/Freescale view these instructions as using an "Implied Register" addressing mode.
- Possible implied CPU registers:
  PC, SSP, USP, SR

Eight Broad Classes of ColdFire Instructions

1) Data Movement
2) Integer Arithmetic
3) Logical Operations
4) Shift Operations
5) Bit Manipulation
6) Program Control
7) System Control
8) Cache Maintenance

Note: Not all of the ColdFire instructions are listed in the following pages. Consult the ColdFire Family Programmer's Reference Manual for the full list.

1) Data Movement Instructions

MOVE.s <ea1>,<ea2>	General-purpose data movement, s = B, W or L
MOVEA.s <ea>,Ax	Move operand into an address register, s = W, L
MOVEM.L <ea1>,<ea2>	Move multiple registers to/from memory region
MOVEQ.L #<data>,Dx	Move 8-bit data to data register
MOVE.W CCR,Dx	Move zero-padded CCR to a data register
MOVE.B <ea>,CCR	Move 8-bit data into the CCR
MOV.L <ea1>,<ea2>	Special MOVE instructions for the 8 EMAC regs. MACSR, ACC0/1/2/3, ACCext01/23 and MASK
MOVCLR.L ACCn,Rm	Rm <- ACCn in EMAC, then clear ACCn
LEA.L <ea>,Ax	Load effective address (i.e., initialize a pointer)
PEA.L <ea>	Push a pointer onto the A7 stack
LINK Ay,#<disp>	Create a stack frame to hold local variables
UNLK An	Remove a stack frame and restore frame pointer

2) Integer Arithmetic Instructions

ADD.L <ea1>,<ea2>	Add, where one operand must be a data register
ADDA.L <ea>,Ax	Add source operand to an address register
ADDI.L #<data>,Dx	Add 32-bit immediate data to a data register
ADDQ.L #<data>,Dx	Add a value from 1 to 8 to a data register
ADDX.L Dy,Dx	Add Dy + Dx + X bit and then store sum in Dx.L

Subtract instructions are also available, with SUB replacing ADD above

MULU.s <ea>,Dx	Multiply (unsigned) to Dx.L, s = W or L
DIVU.W <ea>,Dx	Divide (unsigned) Dx.L by <ea>.W and store the quotient in Dx.W and remainder in Dx.L[31:16]
DIVU.L <ea>,Dx	Divide (unsigned) Dx.L by <ea>.L and then store the quotient in Dx.L. Discard the remainder.
REMU.L <ea>,Dw:Dx	Find remainder (unsigned) of Dx.L divided by <ea>.L and then store in Dw.L

Signed multiply, divide and remainder instructions are available using MULS, DIVS and REMS, respectively.

2) Integer Arithmetic Instructions (cont'd)

CMP.s <ea>,Dx	Compare with data register, s = B, W or L
CMPA.s <ea>,Ax	Compare with address register, s = W or L
CMPI.s #<data>,Dx	Compare data with data register, s = B, W or L
CLR.s <ea>	Clear, with s = B, W or L
NEG.L Dx	Negate Dm: compute (#0 – Dx.L) and store in Dx.L
NEGX.L Dx	Compute (#0 – Dx.L – X bit) and store in Dx.L
EXT.W Dx	Sign extend Dx.B to Dx.W
EXT.L Dx	Sign extend Dx.W to Dx.L
EXTB.L Dx	Sign extend Dx.B to Dx.L

2) Integer Arithmetic Instructions (cont'd)

Additional arithmetic instructions are available for the enhanced multiply-accumulate (EMAC) unit that is present in the MCF54415:

"Multiply Accumulate"

MAC.W Ry.b,Rx.c<scale>,ACCn

MAC.L Ry,Rx<scale>,ACCn

Ry and Rx are address and/or data registers

Upper (U) or lower (L) word specified by "b" and "c"

Optional <scale> is "<<1" or ">>1" or ""

ACCn is one of ACC0, ACC1, ACC2 or ACC3

ACCn <= ACCn + scaled product of Rn.b x Rx.b

"Multiply Accumulate with Load" (same as MAC, but also do Rw.L <= <ea>)

MAC.s Rn.b,Rm.c<scale>,<ea>,Rw,ACCx

MAC.s Rn.b,Rm.c<scale>,<ea>&,Rw,ACCx (also AND <ea> with MASK)

"Multiply and subtract scaled product from accumulator"

Uses same syntax as MAC, but with MSAC instead

3) Bit-wise Logical Instructions

AND.L <ea>,Dx	Bitwise AND of <ea> and Dx, then store in Dx
AND.L Dy,<ea>	Bitwise AND of Dy and <ea>, then store in <ea>
ANDI.L #<data>,Dx	Bitwise AND of immed. data and Dx
OR.L <ea>,Dx	Bitwise OR of <ea> and Dx, then store in Dx
OR.L Dy,<ea>	Bitwise OR of Dy and <ea>, then store in <ea>
ORI.L #<data>,Dx	Bitwise OR of immed. data and Dx
EOR.L <ea>,Dx	Bitwise XOR of <ea> and Dx, then store in Dx
EOR.L Dy,<ea>	Bitwise XOR of Dy and <ea>, then store in <ea>
EORI.L #<data>,Dx	Bitwise XOR of immed. data and Dx
NOT.L Dx	Bitwise negate/flip contents of data register Dx

4) Shift Operations

ASL.L Dy,Dx	Arithmetic shift left; 0's inserted at the LSB
ASL.L #<data>,Dx	Arithmetic shift left; 0's inserted at the LSB
ASR.L Dy,Dx	Arithmetic shift right; sign extended at the MSB
ASR.L #<data>,Dx	Arithmetic shift right; sign extended at the MSB
LSL.L Dy,Dx	Logical shift left; 0's inserted at the LSB
LSL.L #<data>,Dx	Logical shift left; 0's inserted at the LSB
LSR.L Dy,Dx	Logical shift right; 0's inserted at the MSB
LSR.L #<data>,Dx	Logical shift right; 0's inserted at the MSB
SWAP.W Dx	Swap the two 16-bit words in a 32-bit register

Note: The M68000 and CPU32 rotate instructions (e.g., ROR, ROL) are not provided in the ColdFire instruction set.

5) Bit Manipulation Instructions

BSET.s Dy,<ea>	Set bit in <ea> at bit position given in Dy, s = B,L
BSET.s #<data>,<ea>	Set bit in <ea> at bit position #<data>, s = B,L
BCLR.s Dy,<ea>	Clear bit in <ea> at bit position given in Dy, s = B,L
BCLR.s #<data>,<ea>	Clear bit in <ea> at bit position #<data>, s = B,L
BCHG.s Dy,<ea>	Flip bit in <ea> at bit position given in Dy, s = B,L
BCHG.s #<data>,<ea>	Flip bit in <ea> at bit position #<data>, s = B,L
BTST.s Dy,<ea>	(Z bit in CCR) <= complement of specified bit
BTST.s #<data>,<ea>	(Z bit in CCR) <= complement of specified bit
BITREV.L Dx	Bitwise reverse the contents of Dx.L
BYTEREV.L Dx	Byte-wise reverse the contents of Dx.L
FF1.L Dx	Load Dx.L with offset to first 1 bit from MSB position

6) Program Control Instructions

Returns:

RTS	Return from subroutine
RTE	Return from exception handling routine (*Privileged*)

Unconditionals:

BRA <label>	Branch always using 8 or 16-bit displacement
BSR <label>	Branch to subroutine using 8 or 16-bit displacement
JMP <label>	Jump to instruction at <label> in program
JMP <ea>	Jump to instruction at <ea>
JSR <label>	Jump to subroutine at <label> in program
JSR <ea>	Jump to subroutine at <ea>
NOP	No operation (safely waste 3 core clock cycles of time)
TPF	Two-word NOP, with no pipeline synchronization
TPF.W #<data>	Four-word NOP, with no pipeline synchronization
TPF.L #<data>	Six-word NOP, with no pipeline synchronization

6) Program Control Instructions (cont'd)

Test Operand:

TST.s <ea>	Update CCR bits Z and N according to <ea>; s = B,W,L
TAS.B <ea>	Update CCR bits Z and N according to the byte at <ea>. After that update, TAS.B sets bit #1 of the byte at <ea>. *Important*: TAS.B performs an indivisible read-modify-write operation on the bus: ideal for flag & semaphore updates.

Conditionals:

Bcc.B <label>	Branch using 8-bit displacement if condition "cc" is true; otherwise, continue with the next instruction
Bcc.W <label>	Branch using 16-bit displacement if condition "cc" is true; otherwise, continue with the next instruction
Scc.B Dx	Set Dx to 0x1 if condition "cc" is true; otherwise, clear to 0x0

The 16 possible values of the condition "cc" are given on the next slide.

6) Program Control Instructions (cont'd)

The 16 possible values of the condition "cc" are:

T	true	1	
F	false	0	
EQ	equal	Z=1	
NE	not equal	Z=0	
PL	plus	N=0	
MI	minus	N=1	
CC	carry clear	C=0	(HS, high or same, unsigned operands)
CS	carry set	C=1	(LO, low, unsigned operands)
VC	overflow clear	V=0	
VS	overflow set	V=1	
LT	less than	Destn. < Source	(signed operands)
GT	greater than	Destn. > Source	(signed operands)
LE	less than or equal	Destn. <= Source	(signed operands)
GE	greater than or equal	Destn. >= Source	(signed operands)
LS	low or same	Destn. <= Source	(unsigned operands)
HI	high	Destn. > Source	(unsigned operands)

7) System Control Instructions

Moves involving the Condition Code Register (non-privileged):

MOVE.B CCR,Dx	
MOVE.B Dy,CCR	*Note*: Upper byte of SR is not changed
MOVE.B <ea>,CCR	*Note*: Upper byte of SR is not changed

Privileged instructions (supervisor mode only):

MOVE.W SR,Dx	*Note*: Unimplemented bits read as 0's
MOVE.W Dy,SR	
MOVE.W <ea>,SR	
MOVEC.L Ry,Rx	Move longword between control registers
STRLDSR #<data>	Push SR onto stack, and then SR <= #<data>
RTE	Return from exception handling routine
HALT	Halt processor core
STOP #<newSR>	Load SR; stop execution and wait for interrupt

7) System Control Instructions (cont'd)

Trap-generating:

TRAP #<4bitval>	Force one of 16 possible traps
ILLEGAL	Force an ILLEGAL exception

Debug:

PULSE	Sends code to Processor Status (PST) output port
WDDATA.s <ea>	Drive debug data to DDATA output port, s = B,W,L
WDEBUG.L <ea>	(Privileged) Execute loaded debug command

8) Cache Maintenance Instructions

CPUSHL "Push and possibly invalidate cache line"
If data is valid and modified, push specified cache line; invalidate cache line if programmed in Cache Control Register (CACR) (this synchronizes the CPU pipeline).

The MCF54415 contains an 8-Kbyte instruction cache memory to speed up instruction execution. The cache can be configured using the CACR into the following modes:

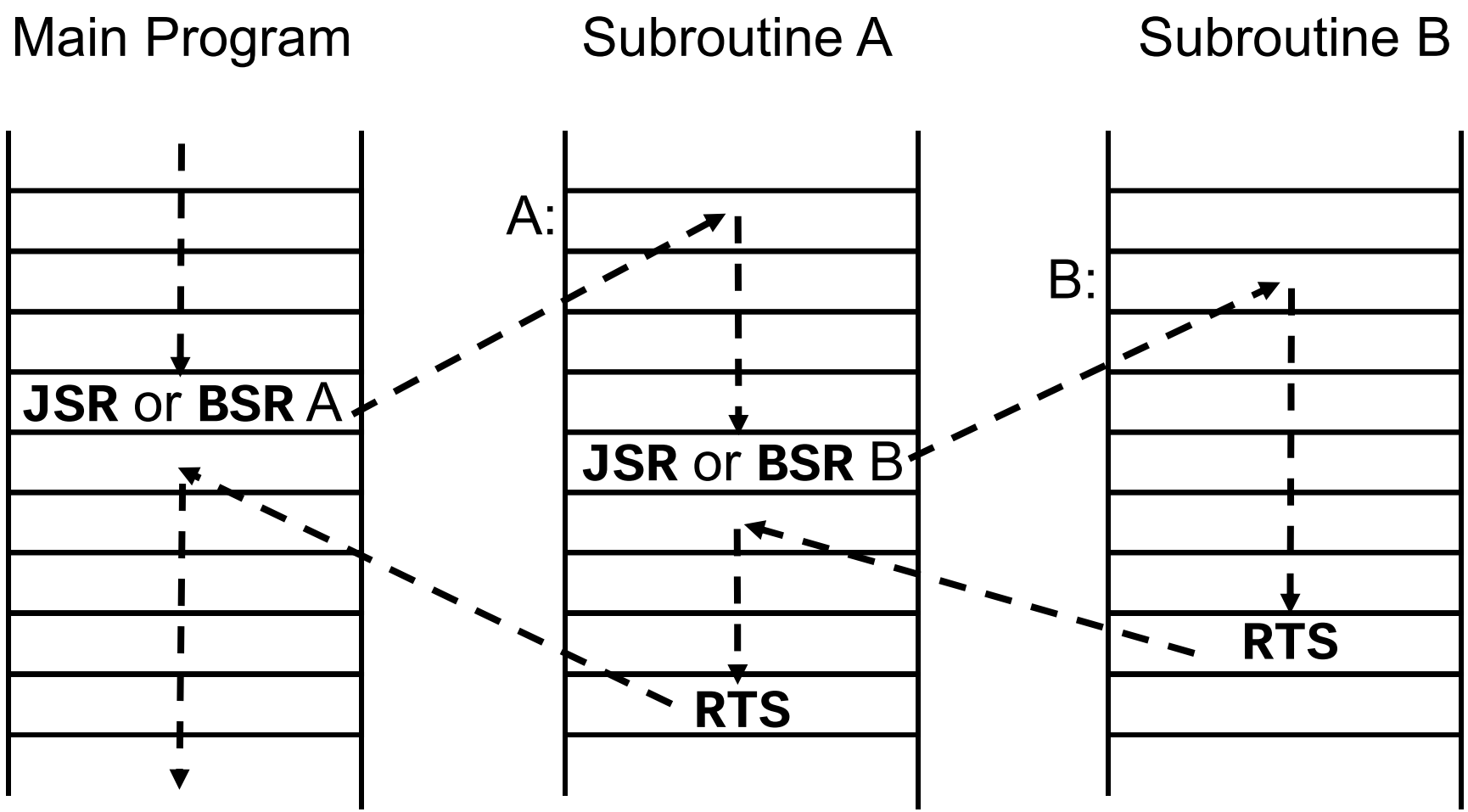
- Instruction cache
- Write-through data cache
- Split instruction/data cache

CPUSHL is a way of invalidating a single cached instruction. It is faster than starting a full cache invalidation sequence. CPUSHL is a privileged instruction.

Subroutines

- A subroutine is a short program segment that can be called by any other program, including itself.
- Subroutines are typically used to encapsulate frequently used code.
- Subroutines provide more efficient use of memory since the enclosed code does not have to be repeated in several places in other program(s).
- Subroutines promote the re-use of known-good code, which increases programmer productivity.
- Subroutines cost a small extra amount of execution time when compared to “in-line” code.

Execution Pattern for Subroutine Calls



Parameter Passing

- Input and output parameters can be passed into and out of a subroutine using a number of different techniques:
  - CPU registers
  - memory locations in the heap (not on the stack)
  - memory locations on the stack
  - as values embedded among the instructions

The Normal Processing State

- In normal instruction processing, the next instruction that will be executed by the CPU is the one that is pointed to by the Program Counter (PC) once the end of the current instruction has been reached.

- This PC value is determined either:
  - Implicitly: The PC is incremented automatically past the end of the current instruction to point to the first word of the next instruction in memory.
  - Explicitly: The PC is changed to any desired address. This address may be given as an absolute address (as in jumps), or as a branch displacement that must be added to the implicitly determined PC value.

Exceptions, Traps, and Interrupts

- It is often convenient to have a way of diverting the CPU out of the normal processing state so that it can deal promptly with special situations. Freescale (following Motorola) calls these situations “exceptions” and “interrupts”.

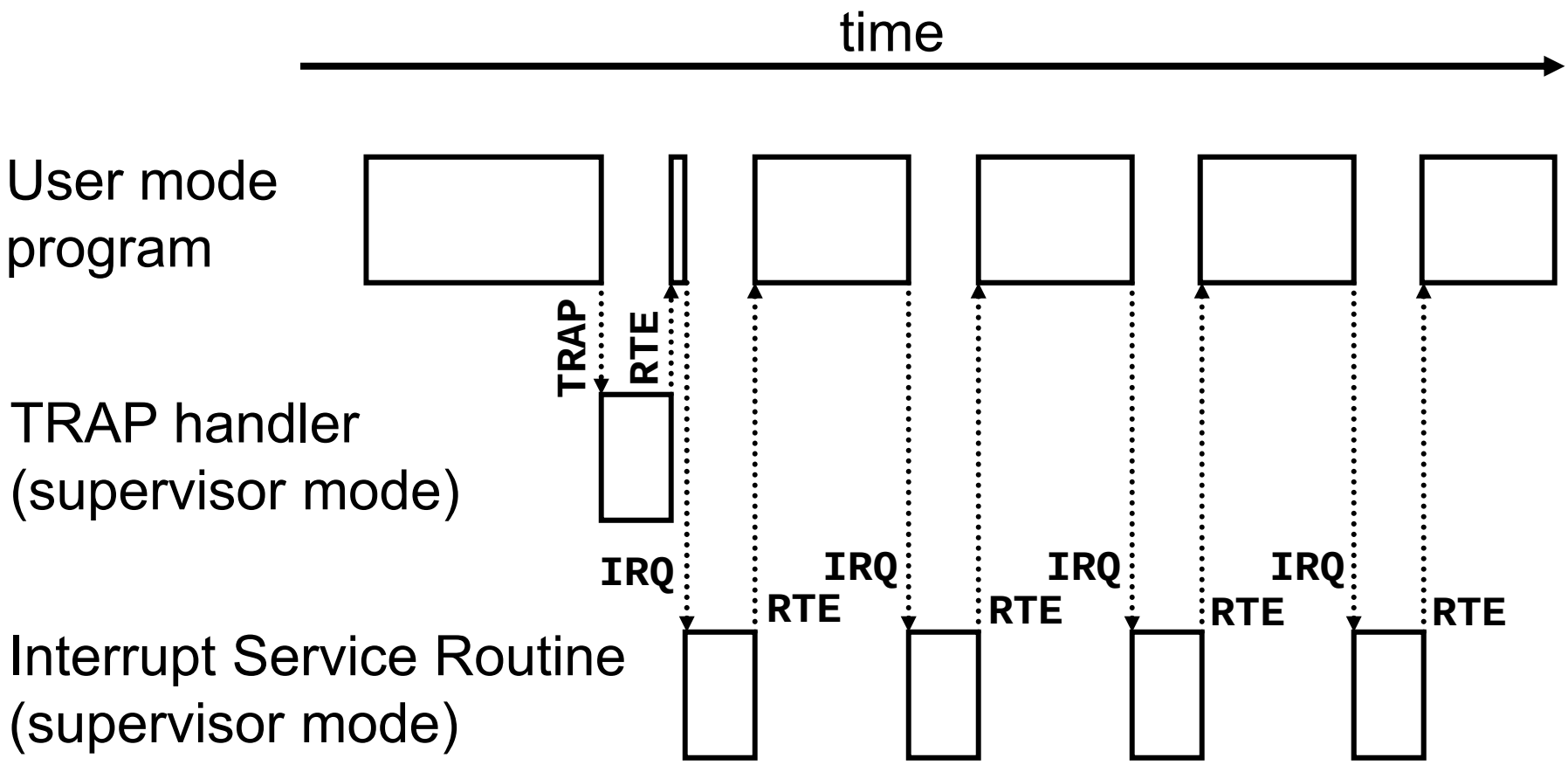
  1) Typical internally-caused exceptions (also called traps):
  - software traps: e.g., I/O routines, OS routines
  - arithmetic exceptions: e.g. divide-by-zero
  - trace mode for program debugging

  2) Typical externally-caused exceptions (also called interrupts):
  - hardware interrupts: e.g. I/O hardware, timers
  - bus error
  - hardware reset signal

Exception Processing (cont'd)

- Exceptions are dealt with by subroutine-like programs called "exception handling routines". Sometimes we will also refer to "trap handlers" and "interrupt service routines". These routines all execute in Supervisor Mode.
- Exceptions can occur during both normal program instruction execution, as well as during the execution of exception handling routines.
- The supervisor stack is used to store the return address of the program that was interrupted when the CPU was required to temporarily halt executing the current program, and then start executing the appropriate exception handling routine.
- Once an exception handling routine has finished, execution is returned to the interrupted program (or routine) by the **RTE** (return from exception) instruction.

Interrupt-driven I/O Execution Pattern



Series of IRQ-triggered I/O data transfers

Interrupt Priority Levels

- Interrupt exceptions can be generated by events in peripheral devices (via peripheral interface chips) in the microcomputer or by other subsystems in the microcontroller chip itself (e.g., FEC, DMACs, timers in the MCF54415).
- The designer of the microcomputer must have grouped the possible interrupts into seven Interrupt Priority Levels.
- The "Interrupt Priority Mask" (I3-I0) in the SR records the level of the interrupt currently being handled. If no interrupt is currently active, then I3-I0 = **`0b000`**.

  Level 7: (Highest priority)

  – A non-maskable interrupt: When such an interrupt occurs, the associated exception handling routine will be called even if another exception handling routine is being executed.

  Levels 6-1: (Six maskable levels, in decreasing priority)

  – An interrupt at any of these priorities is serviced by the CPU only if there is no other active interrupt at the same or at a higher priority level.

The Interrupt Mask Bits

Interrupt Mask Value			Levels Disabled
I2	**I1**	**I0**	**(Masked)**
1	1	1	Levels 1 - 6
1	1	0	Levels 1 - 6
1	0	1	Levels 1 - 5
1	0	0	Levels 1 - 4
0	1	1	Levels 1 - 3
0	1	0	Levels 1 - 2
0	0	1	Level 1
0	0	0	All levels enabled

User vs. Autovectored Interrupts

- *User Interrupts*:
  - Initiated by events in peripheral subsystems, such as timers, interface chips, etc.
  - An “exception vector number” that indicates to the CPU where in memory to find the exception handling routine. In the M68000 and CPU32, the external device supplies this number during the IACK cycle.

- *Autovectored Interrupts*:
  - Also initiated by events in peripheral subsystems.
  - Interrupt handling hardware automatically determines which of 7 possible “levels” an active interrupt should be associated with.
  - The resulting level determines one of the 7 available autovectored interrupt handling routines to use.

Exception Vectors

- An “exception vector” contains the long-word (4-byte) starting address of an “exception handling” or “interrupt service” routine in memory.

- The RESET exception vector is actually *two* vectors:
  - The address of the RESET handling routine.
  - The address of the first supervisor stack pointer.

- M68000, CPU32 and ColdFire microprocessors store 256 exception vectors together in an “Exception Vector Table”.
  - The table occupies the first 1024 bytes in the MC68000.
  - The table is pointed to by the contents of the “vector base register (VBR)” in the CPU32 and ColdFire. The table does not have to occupy the first 1024 bytes in memory (more flexibility).
  - In the ColdFire, the lowest 20 bits of the VBR are 0. This forces the Exception Vector Table to be aligned on 1-Mbyte boundaries.

ColdFire Exception Vector Table

Vector Number(s)	Vector Offset (Hex)	Stacked Program Counter	Assignment
0	0x000	—	Initial stack pointer
1	0x004	—	Initial program counter
2	0x008	Fault	Access error
3	0x00C	Fault	Address error
4	0x010	Fault	Illegal instruction
5	0x014	Fault	Divide by zero
6–7	0x018–0x01C	—	Reserved
8	0x020	Fault	Privilege violation
9	0x024	Next	Trace
10	0x028	Fault	Unimplemented line-a opcode
11	0x02C	Fault	Unimplemented line-f opcode
12	0x030	Next	Debug interrupt
13	0x034	—	Reserved
14	0x038	Fault	Format error
15–23	0x03C–0x05C	—	Reserved
24	0x060	Next	Spurious interrupt
25–31	0x064–0x07C	—	Reserved
32–47	0x080–0x0BC	Next	Trap # 0-15 instructions
48–63	0x0C0–0x0FC	—	Reserved
64–255	0x100–0x3FC	Next	User-defined interrupts

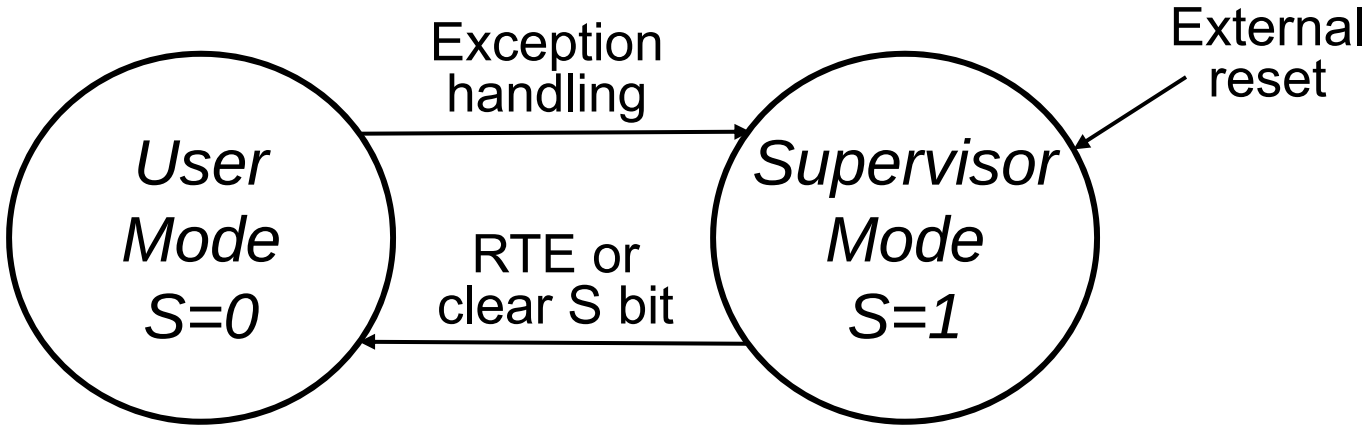
"Fault" refers to the PC of the instruction that caused the exception; "Next" refers to the PC of the next instruction that follows the instruction that caused the fault.

Interrupt/Exception Priorities

- The MCF54415 contains three interrupt controller modules, denoted by INTC0 (highest priority), INTC1 and INTC2 (lowest priority).
- Each INTCx can handle up to 64 interrupt/exception sources
  - 7 external sources (up to 5 of them can be interrupt signals)
  - 173 fully-programmable interrupt/exception sources altogether
- The *Interrupt Acknowledge* (IACK) cycle is handled inside the MCF54415 by the two interrupt controllers (an IACK cycle does not appear on the external bus, as it did in the M68000 and CPU32). Interrupt requests and exceptions must be “cleared” in the hardware by the Interrupt Service and Exception Handling Routines.
- All exceptions after the first 63 in the Exception Vector Table are organized into 7 priority “Levels”, with each level containing 9 “priorities”.
- Within each level, the one fixed-level interrupt source has a midpoint priority. Eight additional interrupt sources can be mapped in software to four higher-than-midpoint and four lower-than-midpoint priorities.

Transitions Between User and Supervisor States

- The handling of an exception causes a transition to Supervisor Mode, which is the mode used by all exception handling routines.
- Exception handling routines usually end with an **RTE** instruction, which restores the pre-existing mode.
- A supervisor mode program can also clear the S bit to enter User Mode. This can be done with a **MOVE** to **SR**.



Automatic Exception Processing Sequence

Step 1: (Save the old SR, and change the processor state)

- Make an internal copy of the SR.
- Enter Supervisor Mode by setting the S bit in the SR.
- Disable tracing by clearing the T bit(s). There is only one T bit in the 68000 and ColdFire; there are two T bits in the CPU32.
- For interrupts of a sufficiently high priority and for reset exceptions, update the interrupt priority mask bits (I2, I1, I0) in the status register. In the ColdFire, also clear the M bit (Master/Interrupt) in the SR.

Step 2: (Get the appropriate Exception Vector Number)

- For interrupts, get the appropriate exception vector number during an “Interrupt Acknowledge Cycle” (IACK). During an IACK cycle, the interrupt priority is placed on the address bus.
- For all other exceptions, use internal CPU logic to determine the corresponding 8-bit Exception Vector Number.

Automatic Exception Processing Sequence

Step 3: (Save the current processor state on the stack)

- Create an exception stack frame on top of the supervisor stack.
- Load the SR and PC values, and possibly other information, into the new stack frame.
- The ColdFire has a single exception frame format.
- In the M68000 and CPU32, there were several frame formats.

Step 4: (Start executing the exception handling routine)

- Multiply the 8-bit exception vector number by four to get the offset of the 32-bit Exception Vector into the vector table.
- Go into the exception vector table (in the M68000 the table base address is 0x0; in the CPU32 and ColdFire, the base address is in the VBR), retrieve the Exception Vector, and load it into the PC.
- If no other higher-priority exception is pending, resume the normal instruction fetch-decode-execute cycle.
- In the ColdFire, interrupt handling is disabled for the next instruction so that STRLDSR can be used to change the interrupt mask.

The Reset Exception

- A reset exception is caused by a hardware reset signal originating from outside the microcontroller. (The ColdFire, unlike the 68xxx and CPU32, does not have a software RESET instruction for producing reset exceptions.)

- The reset exception is used for system initialization and recovery from catastrophic failure. No CPU state information is saved on the supervisor stack.
  - S bit is set
  - Trace mode is disabled (T bits cleared)
  - M bit is cleared
  - Interrupt Priority Mask is set to **`0b111`**
  - Vector Base Register (VBR) is cleared
  - SP is loaded with vector 0 at address **`$000`**
  - PC is loaded with vector 1 at address **`$004`**
  - First instruction of the system initialization routine is fetched, decoded and executed

Memory Map Organization in the MCF54415

Table 1-2. System Memory Map per Boot Mode

Address Range[1]	Boot Source		
	FlexBus	Flash Controller	Serial Boot
0x0000_0000 0x0000_FFFF	FlexBus	NAND flash controller[2]	Internal SRAM[2]
0x0001_0000 0x00FF_FFFF			FlexBus
0x1000_0000 0x3FFF_FFFF		FlexBus	
0x4000_0000 0x7FFF_FFFF	SDRAM controller		
0x8000_0000 0x8BFF_FFFF[3]	Internal SRAM backdoor[3]		
0x8C00_0000 0x8FFF_FFFF	Rapid GPIO		
0x9000_0000 0xBFFF_FFFF	Reserved		
0xC000_0000 0xDFFF_FFFF	FlexBus		
0xE000_0000 0xEFFF_FFFF	Peripheral bus controller 1[4]		
0xF000_0000 0xFFFF_FFFF	Peripheral bus controller 0[4]		

Boot software for bootstrapping the system: i.e., hardware configuration and testing, loading the operating system, etc.

1 Gbytes of SDRAM for the operating system, application software, and application data

Fast (on-chip) 64-Kbyte SRAM is mapped into this range. Locations can be accessed by the CPU and by one other bus master at the same time.

Noncacheable external peripherals

Noncacheable internal peripherals

MCF5441x Reference Manual, Section 1.8