Mikroprocesorová technika v embedded systémech

Ing. Jaroslav Lepka
Ing. Libor Prokop

27. – 28. května 2010

Tato prezentace je spolufinancována Evropským sociálním fondem a státním rozpočtem České republiky.

Časový rozvrh

- 09:00 – 10:30 – blok 1
- 10:30 – 10:45 – přestávka
- 10:45 – 12:00 – blok 2
- 12:00 – 13:00 – oběd
- 13:00 – 14:15 – blok 3
- 14:15 – 14:30 – přestávka
- 14:30 – 15:45 – blok 4
- 15:45 – 16:00 – přestávka
- 16:00 – 17:00 – blok 5

General Information

RISC, CISC, DSP

CISC Processor Architecture

- CISC – Complex Instruction Set Computing
- Example of CISC processors – Intel 80x86, Motorola 68K
- CISC - typical characteristics
  - Easy to program
  - Efficient use of memory
  - Variable length instructions – length often varies according to the addressing mode
  - Instructions require multiple clock cycles to execute
  - Multiple addressing modes for memory
  - Complex instruction-decoding logic
  - Small number of general purpose registers
- Changes in the software and hardware technology caused that modern CISC processors are hybrid implementing many RISC principles
- Make a compiler development simpler
- Pentium is considered a modern CISC processor

RISC Processor Architecture

- RISC – Reduced Instruction Set Computer
- Idea – simplified instructions can provide higher performance in case of much faster execution of each instruction
- Example of RISC processors – AMD 29k, ARM, Atmel AVR, PowerPC
- RISC - typical characteristics
  - Reduced instruction set - small highly optimized set of instructions
  - Less complex, simple instructions
  - Uniform instruction format – single word, less demand for decoding
  - General purpose registers
  - Simple addressing mode
  - Few data types in hardware
  - Less transistors dedicated to the core logic
  - Pipelining
- Risc architecture put a grater burden on the software

DSP Processor Architecture

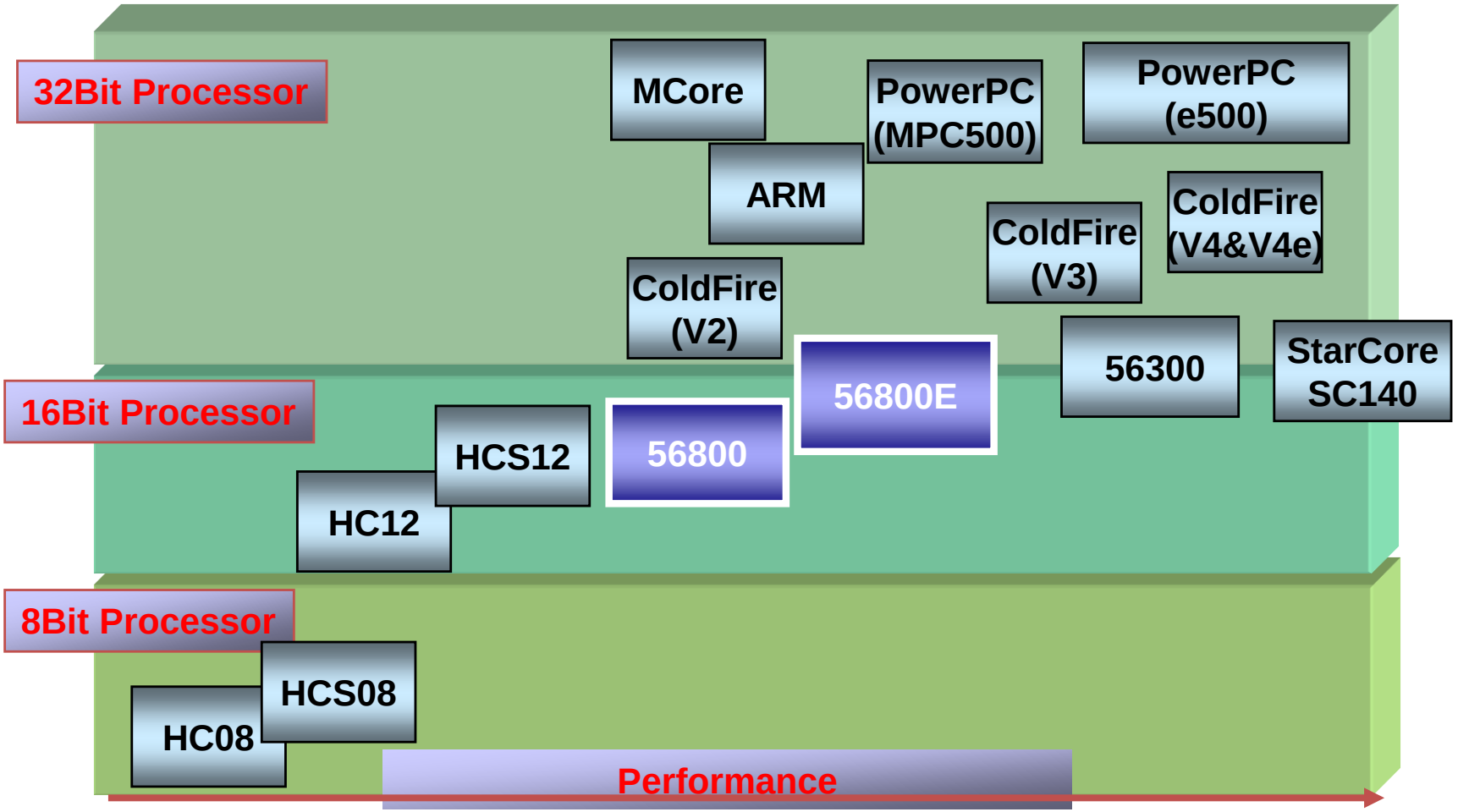
- DSP – Digital Signal Processor
- Suitable for algorithms requiring a large number of mathematical operations to be performed quickly and repetitively
- DSP architecture is optimized for digital signal processing
- DSP - typical characteristics
  - Harvard or dual-Harvard architecture – separate program and data memories
  - Modulo addressing - circular buffers
  - Memory architecture design for streaming data
  - Special arithmetic operations – MAC (Multiply Accumulate), needed for FFT and digital filters processing
  - Special addressing modes
  - Special loop controls – loops without overhead
  - Single-cycle execution of most instructions
  - Lots of special DSP insructions
  - Pipeling

56800E Core Architecture

56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- Address Generation Unit
- Bit Manipulation Unit
- Instruction Pipeline
- Debugging Unit
- 56800E Core Highlights

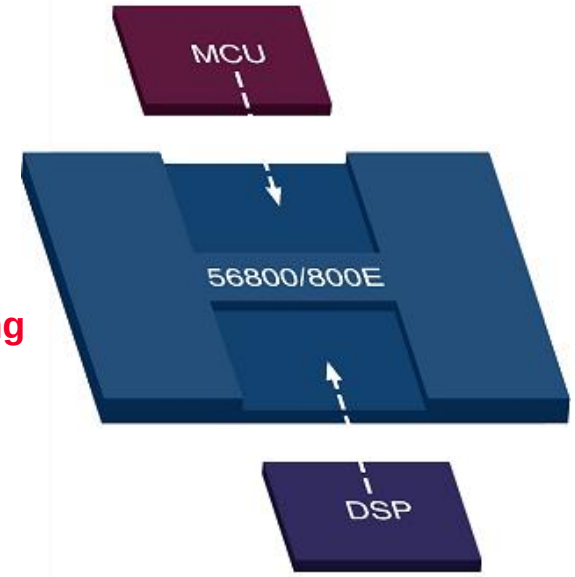
Standard Products Core Roadmap



Freescale DSC: Combining the Best

Traditional Microcontroller

- **Design for Controller Code**
- **Compact Code Size**
- **Easy to Program**
- **Inefficient Signal Processing**



Traditional DSP Engine

- **Designed for DSP Processing**
- **Designed for Matrix Operations**
- **Complex Programming**
- **Less Suitable for Control**

- **Instructions Optimized for Controller Code, DSP, Matrix Operations**
- **Compact Assembly and "C" Compiled Code Size**
- **Easy to Program**
- **Additional MIPS Headroom and extended addressing space**

DSP56800E Core Features

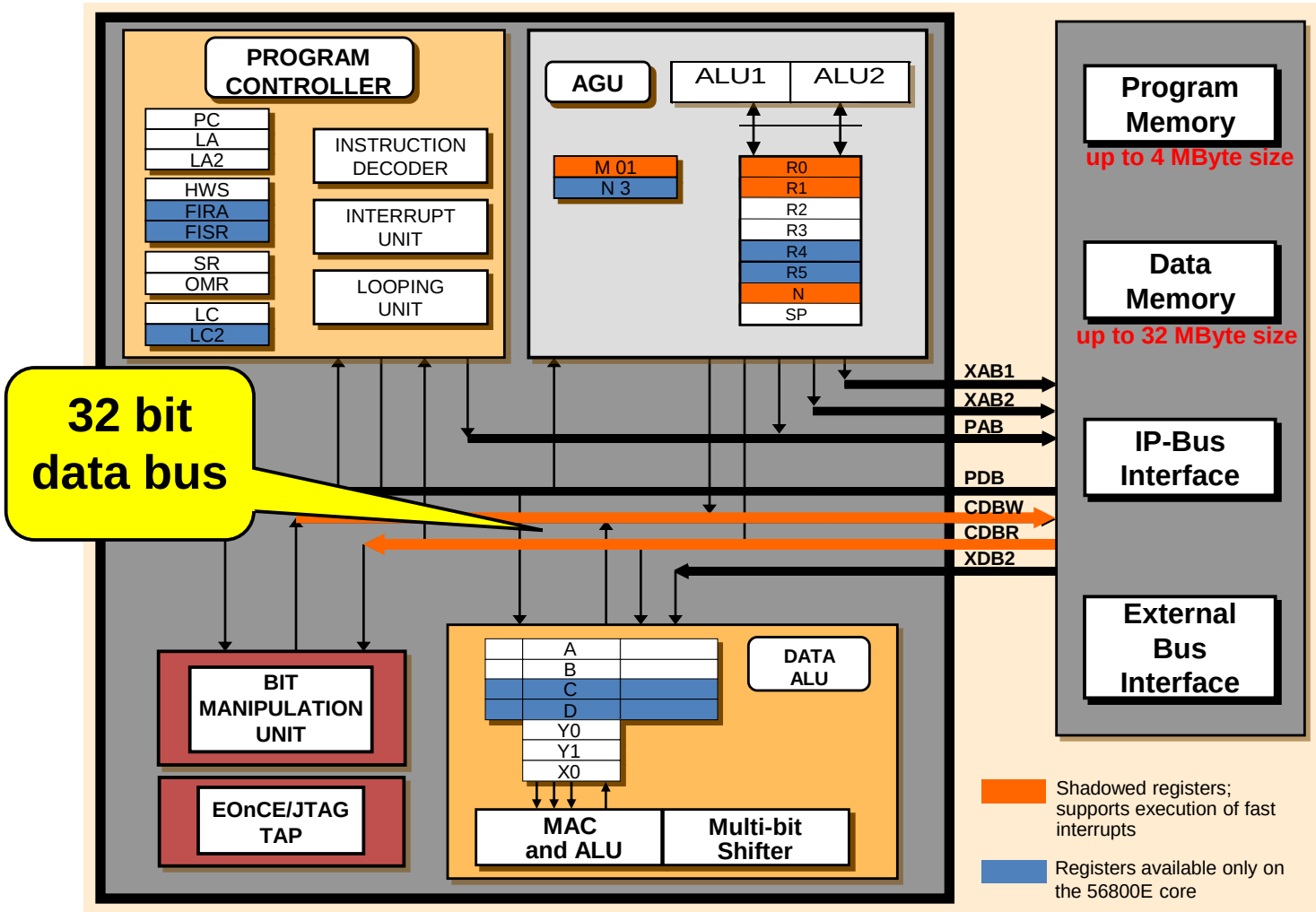
56800/E MCU Functionality

- True Software Stack and Pointer
- 16-bit Program Word
- 20 Addressing Modes and Atomic Read-Modify-Write Instructions
- General Purpose Register Files and Orthogonal Instructions to Data and Address Register Files
- Full Set of Bit and Bitfield Manipulation Instructions and 16- and 32-bit Shifting

56800/E DSP Functionality

- Multiplier - Accumulator (MAC) Single And Dual Parallel Move Instructions
- No Overhead Hardware Looping Nested Looping Capability
- Modulo arithmetic (For Circular Buffers) Integer and Fractional Arithmetic Support
- Nested Interrupt with HW priority Fast Interrupt Support

56800E Hybrid Core Architecture



Instruction Fetch:

PAB - 21 bits
PDB - 16 bits

1st Data Access:

XAB1 - 24 bits
CDBR - 32 bit
CDBW - 32 bit

2nd Data Access:

XAB2 - 24 bits
XDB2 - 16 bits

Operations Performed:

1st - PAB / PDB
2nd - XAB1 / CDBR-CDBW
3rd - XAB2 / XDB2

Address Buses

- Address Buses - three
  - PAB (21-bit) – Program Address Bus (2MWord)
    - Used to address (16-bit) words in program memory
  - XAB1 (24-bit) – Primary Data Address Bus (16MWord)
    - Allows for simultaneous read accesses to data (X) memory
    - Can address byte, word, and long data types
  - XAB2 (24-bit) – Secondary Data Address Bus (16MWord)
    - Allows for simultaneous read accesses to data (X) memory
    - Limited to (16-bit) word accesses

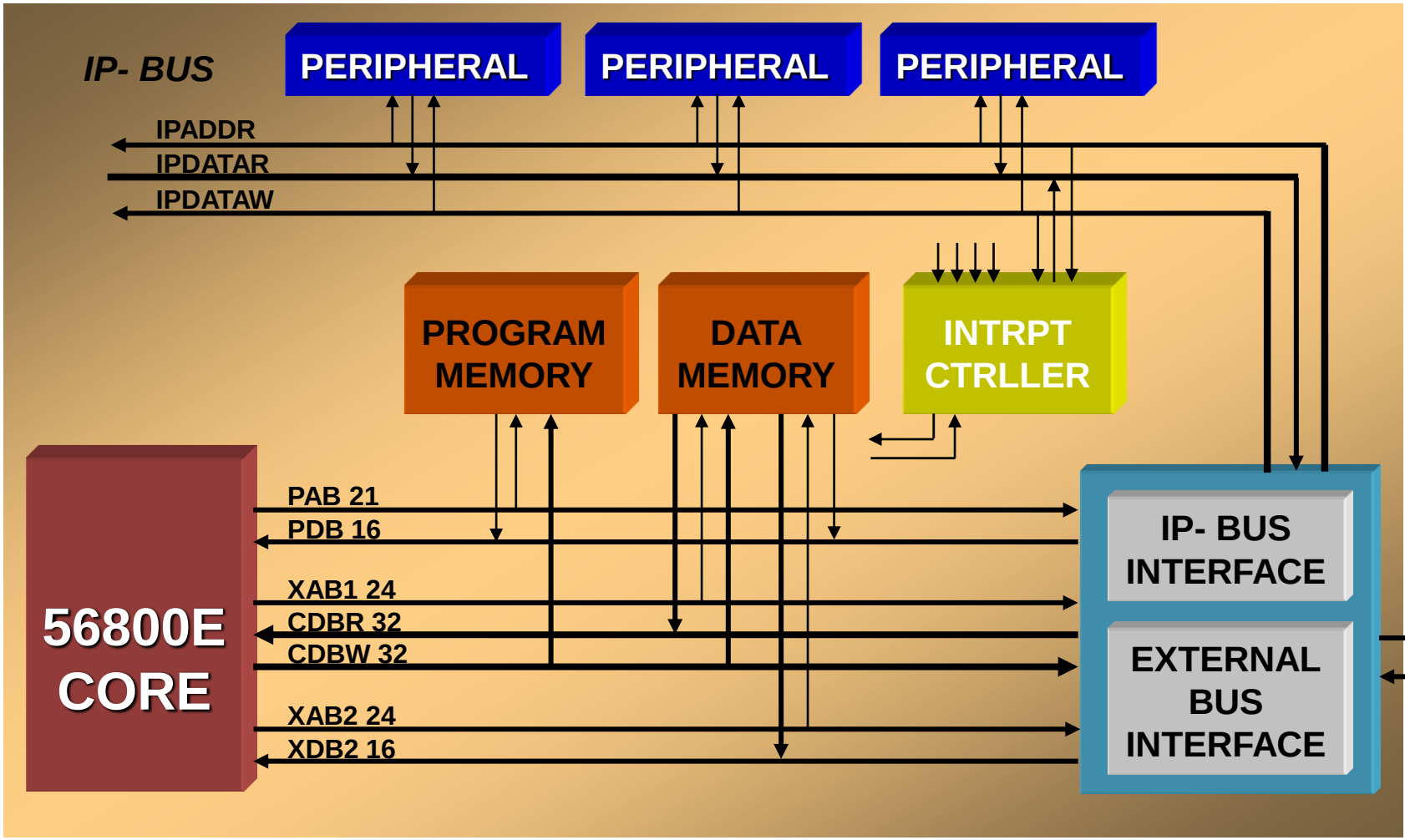
Data Buses

- Data transfers inside the chip occur over the following buses
  - Two uni-directional 32-bit buses
    - Core Data Bus for Reads (CDBR)
    - Core Data Bus for Writes (CDBW)
  - Two uni-directional 16-bit buses
    - Secondary X Data Bus (XDB2)
    - Program Data Bus (PDB)
  - IP-BUS interface

Data Buses

- This bus structure supports up to three simultaneous 16-bit transfers. Any one of the following can occur in a single clock
  - One instruction fetch
  - One read from data memory
  - One write to data memory
  - Two reads from data memory
  - One instruction fetch and one read from data memory
  - One instruction fetch and one write to data memory
  - One instruction fetch and two reads from data memory

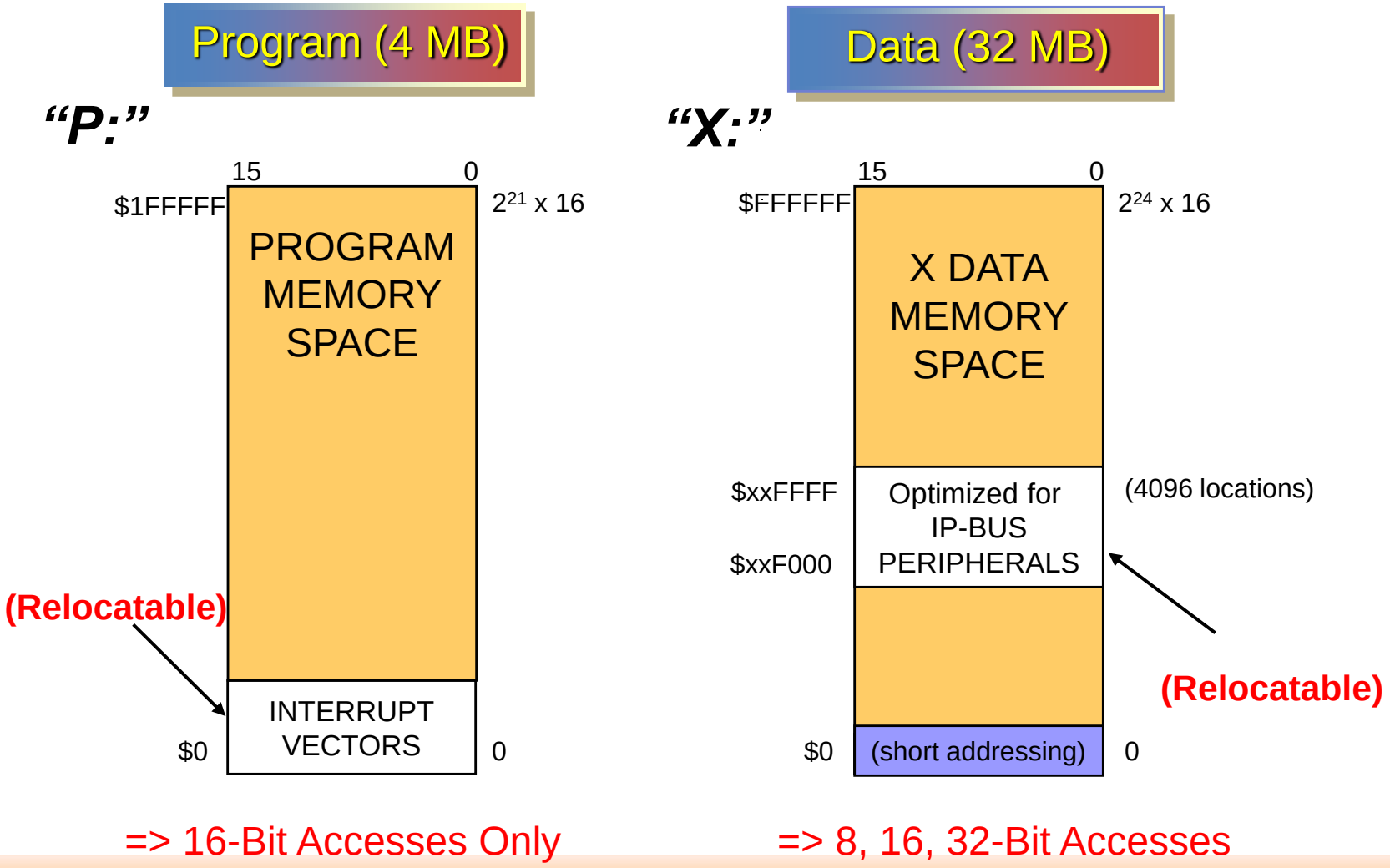
MC56F8300 System Architecture



Dual Harvard Memory Architecture

- The DSP56800E has a dual Harvard architecture with separate program and data memory spaces
- This architecture allows for simultaneous program and data memory accesses
- The data memory interface also supports two simultaneous read operations, enabling up to three simultaneous memory accesses

Program and Data Memories



Programming Model -56800E

DATA ARITHMETIC LOGIC UNIT

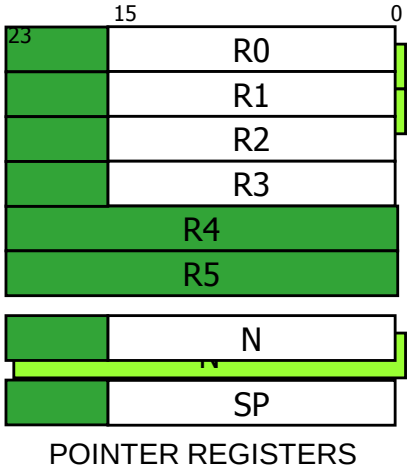
New for 800E

DATA REGISTERS

	35 32	31 16	15 0
A	A2	A1	A0
B	B2	B1	B0
C	C2	C1	C0
D	D2	D1	D0

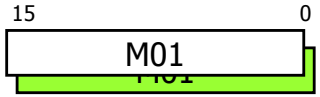
Y	Y1
	Y0
	X0

ADDRESS GENERATION UNIT

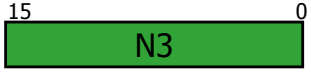


POINTER REGISTERS

==> R0, R1, N, and M01 registers are shadowed

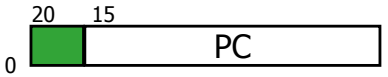


MODIFIER REGISTERS

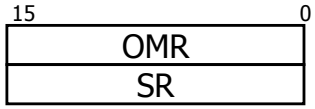


SECONDARY OFFSET REGISTER

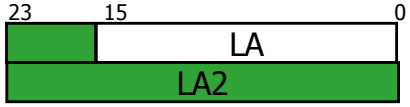
PROGRAM CONTROL UNIT



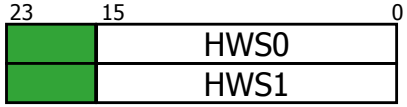
PROGRAM COUNTER



OPERATING MODE and STATUS



LOOP ADDRESS



HARDWARE STACK

LOOP COUNTER

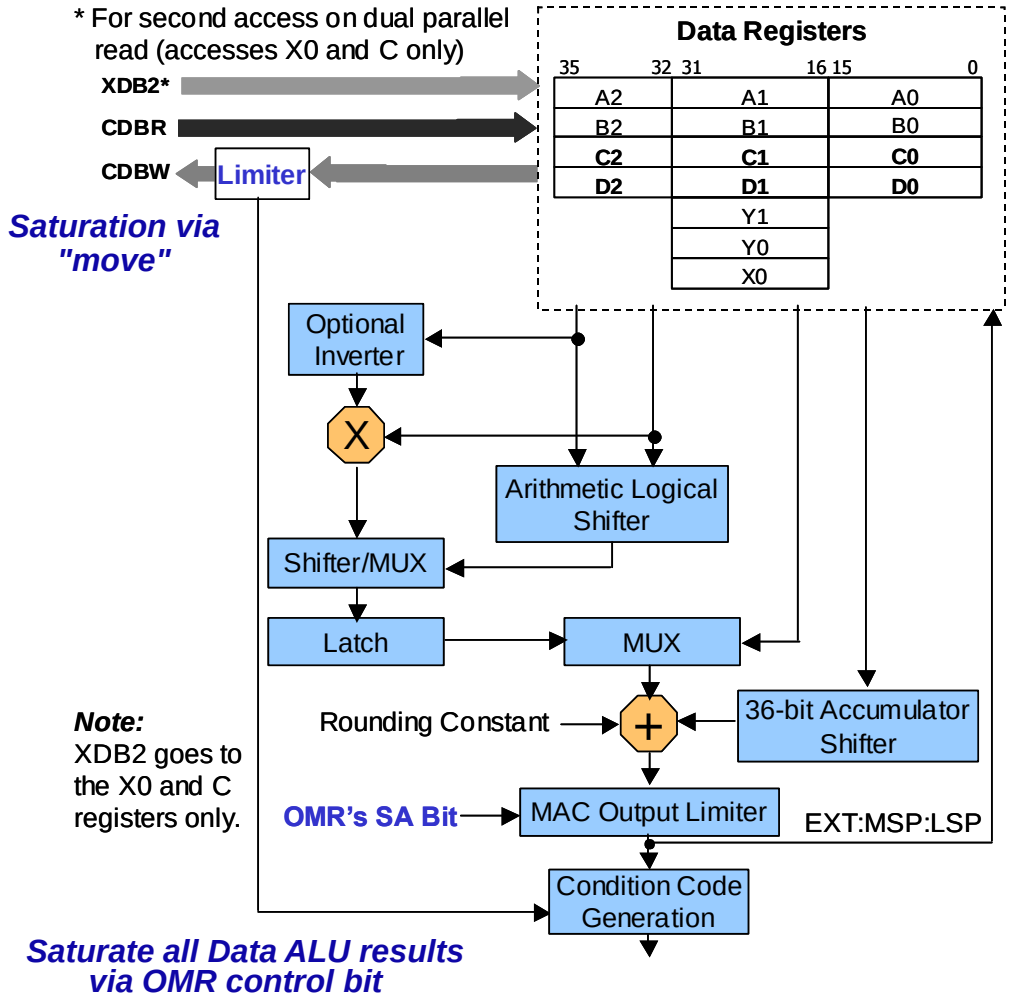
FAST INTERRUPT RETURN ADDRESS

FAST INTERRUPT STATUS REGISTER

56800E Core

- Introduction
- **Arithmetic Logic Unit**
- Program Controller
- Address Generation Unit
- Bit Manipulation Unit
- Instruction Pipeline
- Debugging Unit
- 56800E Core Highlights

Data ALU Block Diagram



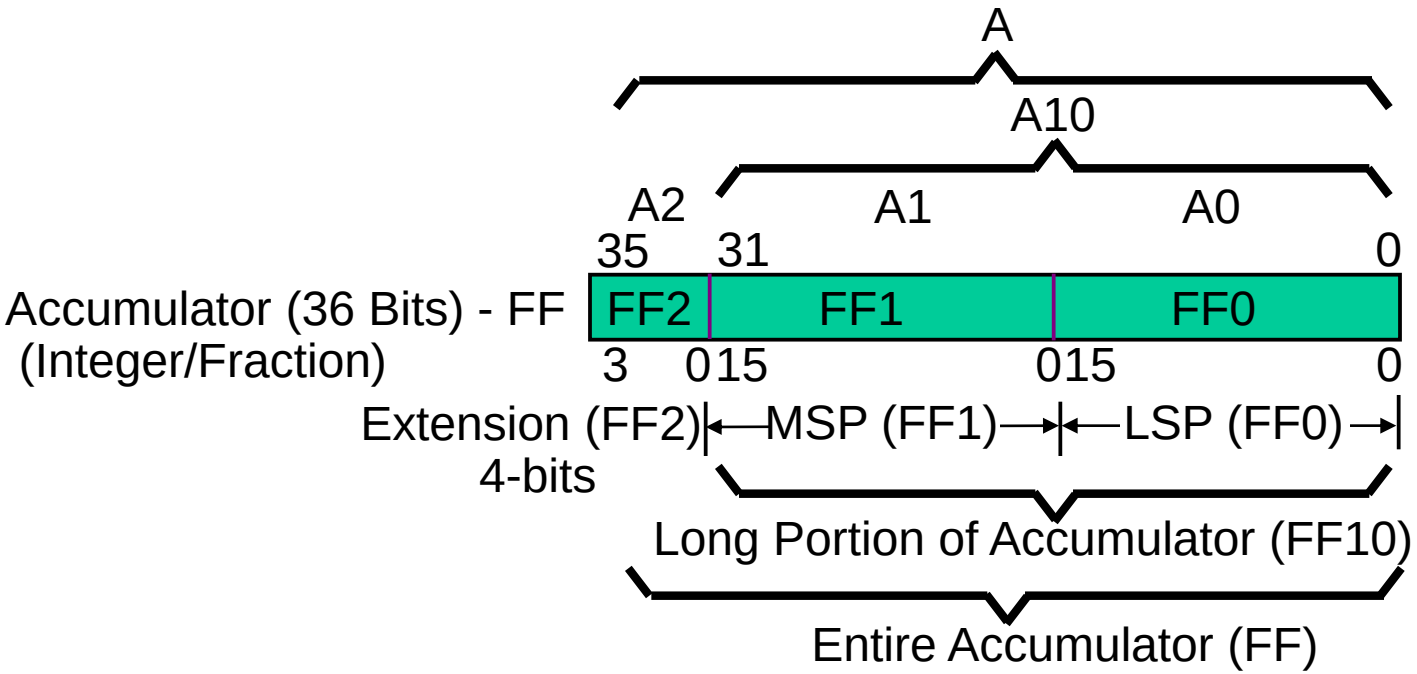
Capabilities

- ✓ Multiplication (w/ Rounding)
- ✓ Multiply-Accumulate (w/ Rounding)
- ✓ Multiprecision Multiplication Support
- ✓ Addition and Subtraction
- ✓ Increments and Decrements
- ✓ Tests and Compares (8, 16, 32, 36 bits)
- ✓ 16 and 32-bit Logical Operations
- ✓ 1's and 2's complement
- ✓ Single Bit Arithmetic & Logical Shifts
- ✓ 16-bit Arithmetic and Logical Shifts
- ✓ 32-bit Arithmetic and Logical Shifts
- ✓ Single Bit Rotates
- ✓ Rounding
- ✓ Absolute Value
- ✓ Sign/Zero Extension
- ✓ Limiting on Move Instructions
- ✓ Conditional Register Transfer
- ✓ Division Iteration Instruction
- ✓ Count Leading Bits
- ✓ Normalization

Accumulator Registers (A, B, C, D)

- Four independent 36-bit accumulator registers
- Each 36-bit data ALU accumulator register is composed of three different portions:
  - 4-bit extension register, FF2 (where FF2 represents A2, B2, C2, or D2)
  - 16-bit most significant product (MSP), FF1 (where FF1 represents A1, B1, C1, or D1)
  - 16-bit least significant product (LSP), FF0 (where FF0 represents A0, B0, C0, or D0)
- Notation
  - FF - refers to the entire accumulator (bits 35–0)
  - FF10 - represents the concatenation of the FF1and FF0 portions and is useful for manipulating 32-bit quantities. (bits 31-0)
  - FF0 - is the 16-bit least significant portion (bits 15–0)
  - FF1 - is the 16-bit most significant portion (bits 31–16)
  - FF2 - refers only to the 4-bit extension portion (bits 35–32)

Accumulator Registers (A, B, C, D)



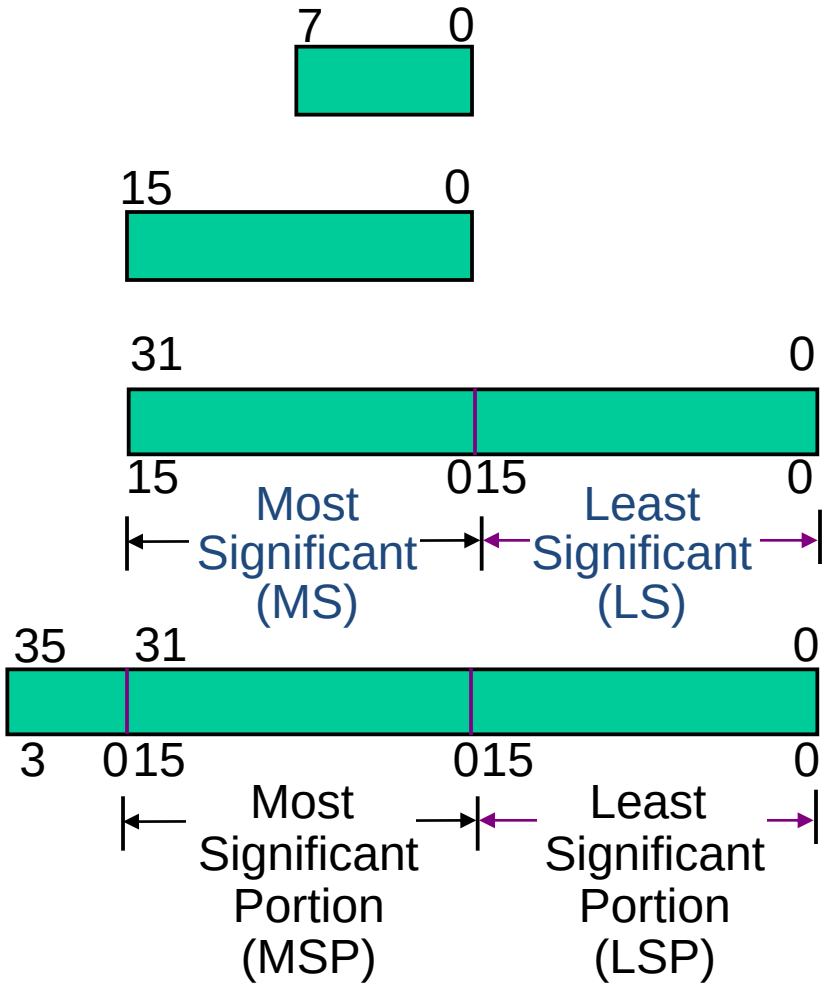
Operand Types

Byte (8 Bits)
(Integer)

Word (16 Bits)
(Integer or Fraction)

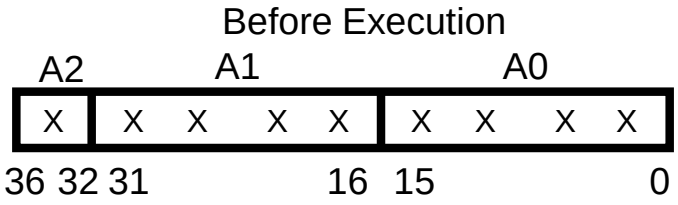
Long word (32 Bits)
(Fraction)

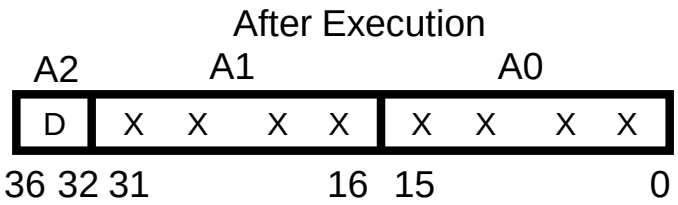
Accumulator (36 Bits)
(Integer/Fraction)



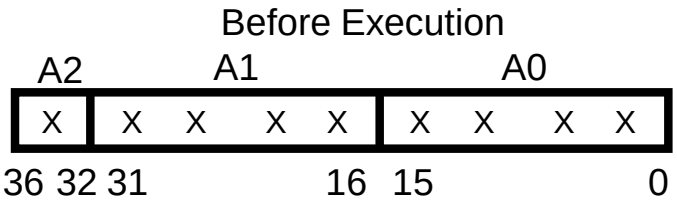
Data Moving

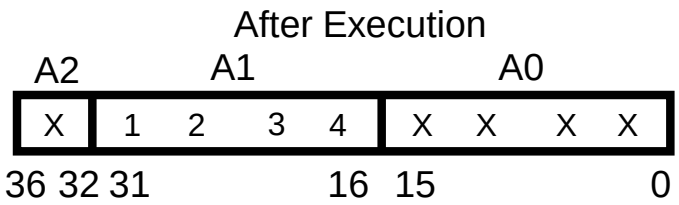
Writing the FF2 portion: **move.w #$ABCD, A2**



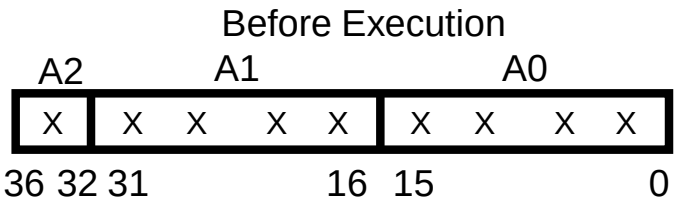


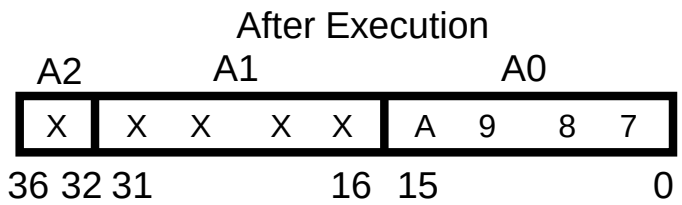
Writing the FF1 portion: **move.w #$1234, A1**





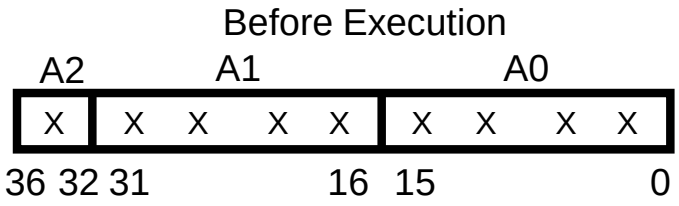
Writing the FF0 portion: **move.w #$A987, A0**

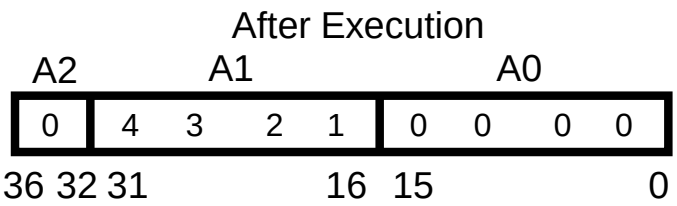




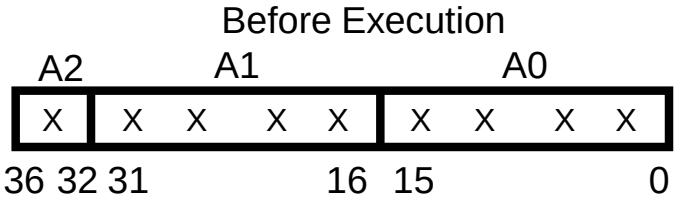
Data Moving

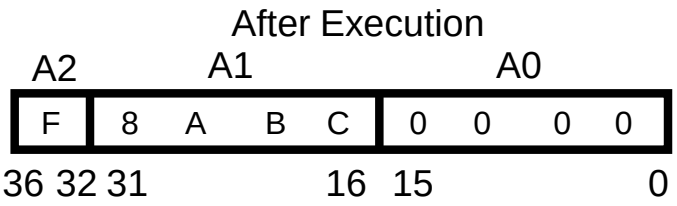
Writing the FF2 portion: **move.w #$4321, A**





Writing the FF1 portion: **move.w #$8ABC, A**

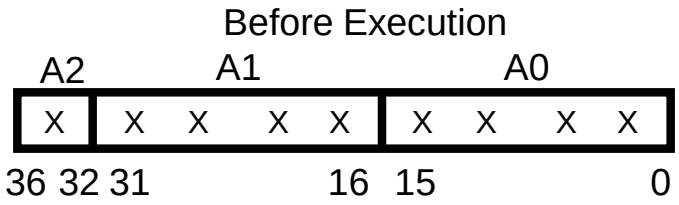


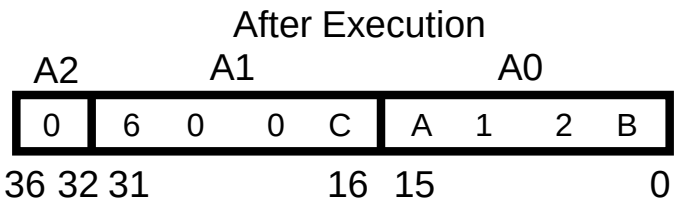


Data Moving

Writing positive immediate value into accumulator:

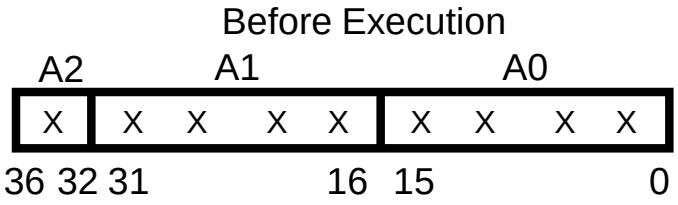
move.L #$600CA12B, A

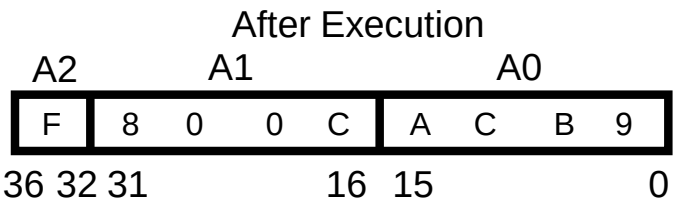




Writing negative immediate value into accumulator:

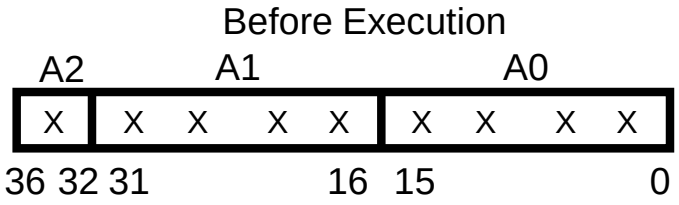
move.L #$800CACB9, A

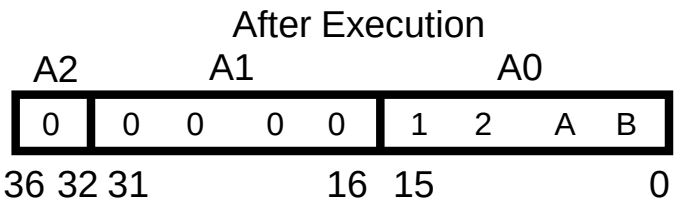




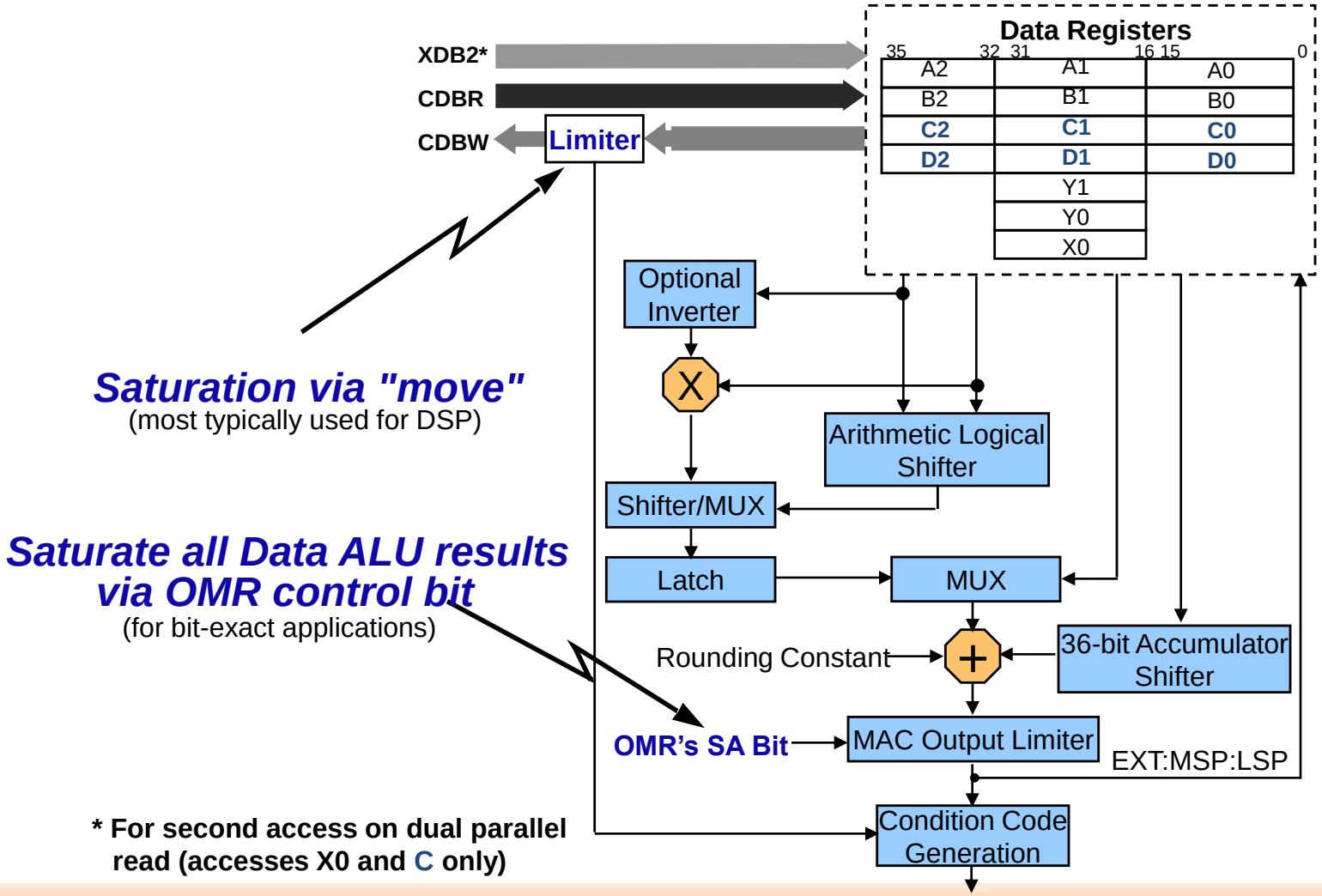
Writing positive immediate value into accumulator:

move.L #$12AB, A





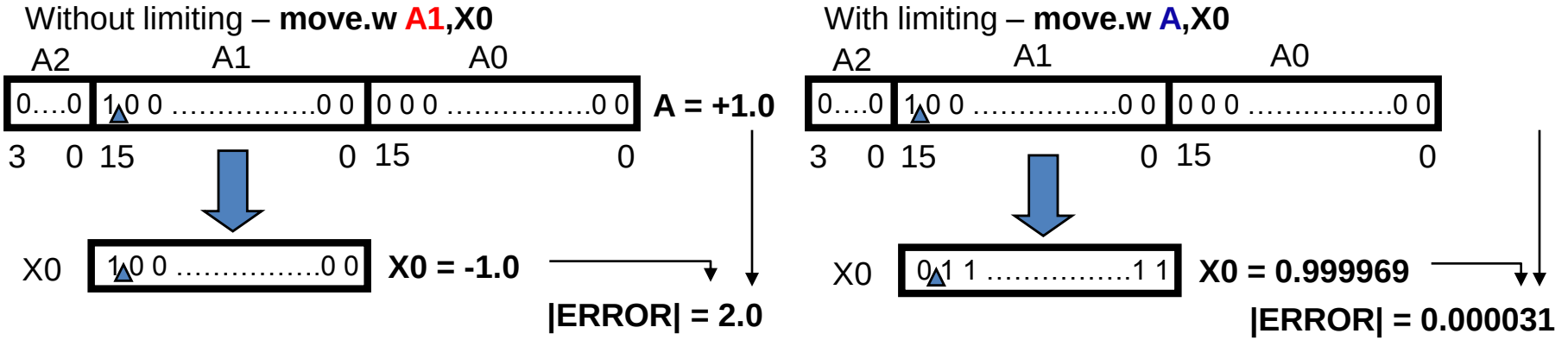
Two Supported Saturation Mechanisms



Data Limiter & MAC Output Limiter

- DSP algorithms can calculate values larger than the data precision of the machine when processing real data streams.
- Normally a processor simply overflows such a result, but this treatment can create problems for processing real-time signals.
- To eliminate the problems associated with overflow and underflow, the DSP56800E provides the optional saturation of results using two limiters:
  - the data limiter
  - the MAC output limiter.

Data Limiting



Demonstrating the Data Limiter—Positive Saturation

```
MOVE.W    #$7FFC,A     ; Initialize A = $0:7FFC:0000
INC.W     A            ; A = $0:7FFD:0000
MOVE.W    A,X:(R0)+    ; Write $7FFD to memory (limiter enabled)
INC.W     A            ; A = $0:7FFE:0000
MOVE.W    A,X:(R0)+    ; Write $7FFE to memory (limiter enabled)
INC.W     A            ; A = $0:7FFF:0000
MOVE.W    A,X:(R0)+    ; Write $7FFF to memory (limiter enabled)
INC.W     A            ; A = $0:8000:0000 <=== Overflows 16 bits!
MOVE.W    A,X:(R0)+    ; Write $7FFF to memory (limiter saturates)
INC.W     A            ; A = $0:8001:0000
MOVE.W    A,X:(R0)+    ; Write $7FFF to memory (limiter saturates)
INC.W     A            ; A = $0:8002:0000
MOVE.W    A,X:(R0)+    ; Write $7FFF to memory (limiter saturates)
MOVE.W    A1,X:(R0)+   ; Write $8002 to memory (limiter disabled)
```

MAC Output Limiter

- The MAC output limiter optionally saturates or limits results that are calculated by data ALU arithmetic operations such as multiplication, addition, incrementing, rounding, and so on
- The MAC output limiter can be enabled by setting the SA bit in the operating mode register (OMR)
- When the SA bit in the OMR is modified, a delay of 2 instruction cycles is necessary before the new saturation mode becomes active

Demonstrating the MAC Output Limiter

```
BFSET     #$0010,OMR     ; Set SA bit-—enables MAC Output Limiter
MOVE.W    #$7FFC,A       ; Initialize A = $0:7FFC:0000
NOP

INC.W     A              ; A = $0:7FFD:0000
INC.W     A              ; A = $0:7FFE:0000
INC.W     A              ; A = $0:7FFF:0000
INC.W     A              ; A = $0:7FFF:FFFF <=== Saturates to 16 bits!
INC.W     A              ; A = $0:7FFF:FFFF <=== Saturates to 16 bits!
ADD.W     #9,A           ; A = $0:7FFF:FFFF <=== Saturates to 16 bits!
```

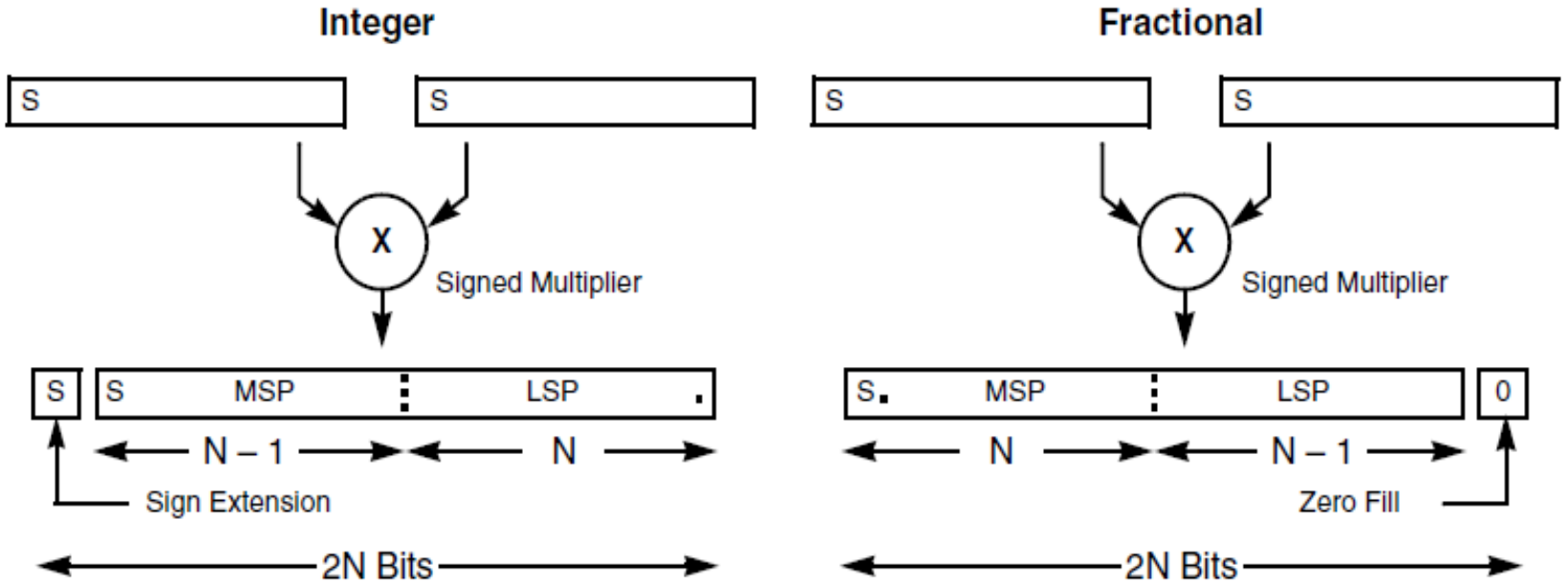
Instructions Not Affected by the MAC Output Limiter

– The MAC output limiter is always disabled (even if the SA bit is set) when the following instructions are executed:

- **ASLL.W, ASRR.W, LSRR.W**
- **ASL16, ASR16, LSR16, ASRAC, LSRAC IMACUU**
- **IMAC.L, IMPY.L, IMPY.W**
- AND.W, OR.W, EOR.W
- LSL.W, LSR.W, ROL.W, ROR.W, ROL.L, ROR.L
- NOT.W, CLB, SUBL
- ADD.B, ADD.BP, SUB.B, SUB.BP
- TST, TST.B, TST.BP, TST.W, TST.L
- **ASLL.L, ASRR.L, LSRR.L**
- **IMPYSU, IMACUS, IMPYUU,**
- **MPYSU, MACSU**
- AND.L, OR.L, EOR.L
- SXT.B, ZXT.B, SXT.L
- ADC, DIV, SBC
- DEC.BP, INC.BP, NEG.BP
- CMP.B, CMP.BP, CMP.L

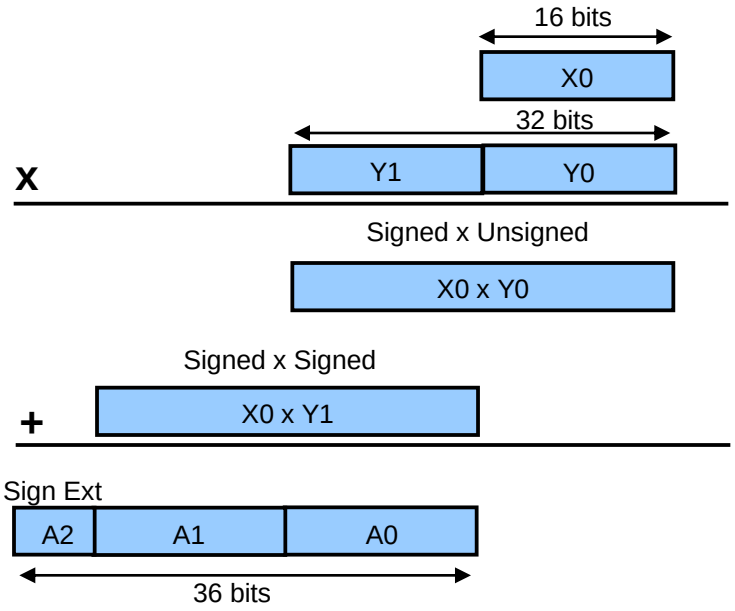
Multiplication

- Signed Multiplication: N x N = 2N - 1 bits



Fractional 16b x 32b => 36b Result

3 Words, 3 Cycles



16 x 32 Bit Fractional Multiplication — 36-Bit Result

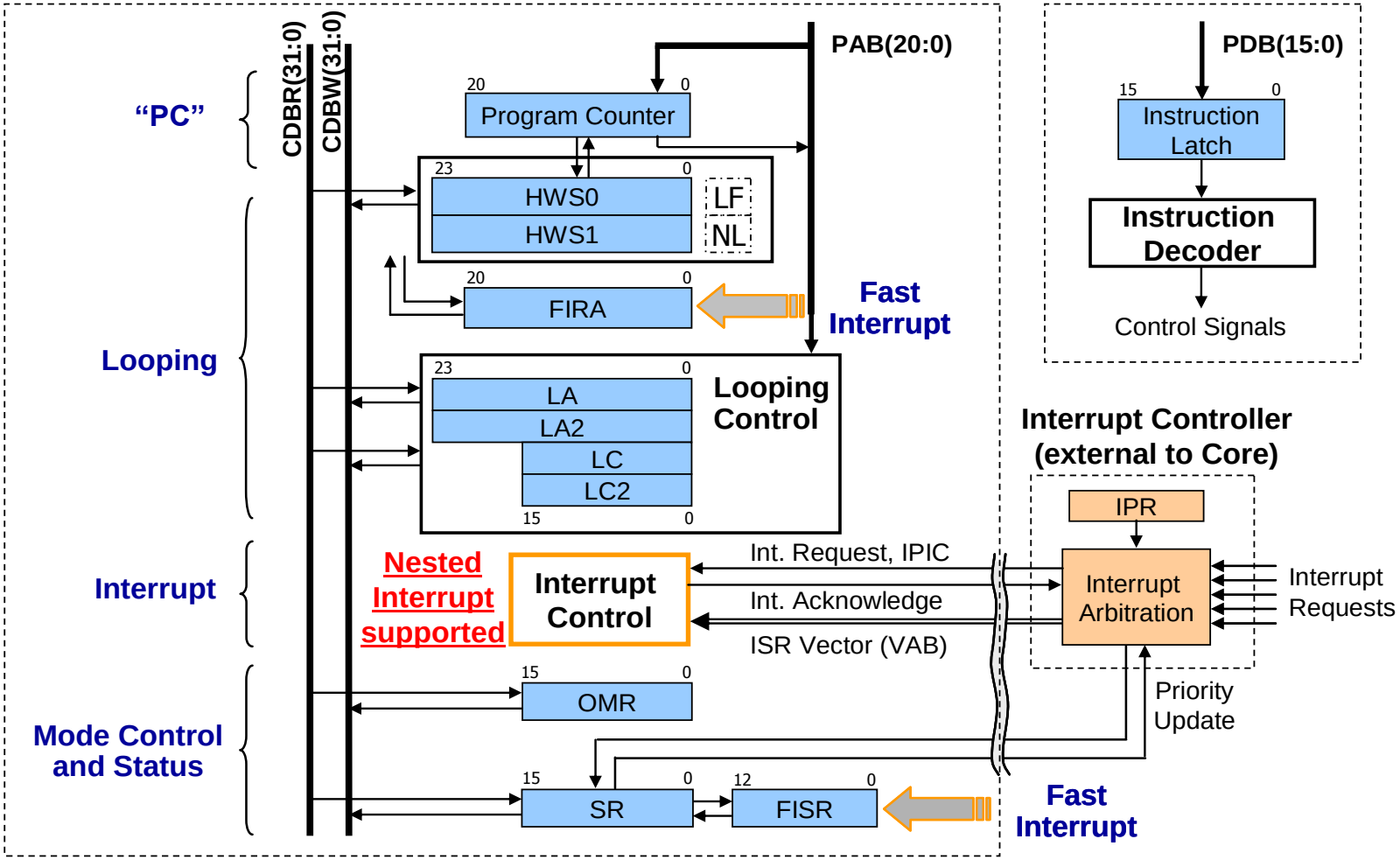
FF2:FF1:FF0 = X0 x Y1:Y0
(Both Fractional Operands are Signed; 3 Cycles, 3 Words)

```
;Signed 16-Bit x Signed 32-Bit Fractional Multiplication
        MPYSU   X0,Y0,A      ; Y1:Y0 = signed X0 x unsigned Y0
        ASR16   A            ; Align 1st product
        MAC     X0,Y1,A      ; A2:A1:A0 = final 36-bit result
```

56800E Core

- Introduction
- Arithmetic Logic Unit
- **Program Controller**
- Address Generation Unit
- Bit Manipulation Unit
- Instruction Pipeline
- Debugging Unit
- 56800E Core Highlights

Program Controller Block Diagram



Nested Hardware Looping

Example - 56800E Assembly Code:

```
    CLR.W    A

    DO     #2,END_OUTR
      DO     #20.END_INNR
        MOVE.W   X:(R0)+,A
        MOVE.W   X:(R1)+,B
        ADD          B,A
        MOVE.W   A,X:(R3)+
END_INNR
      MOVE.W   A,X:(R5)+

END_OUTR
```

Theoretical Best:
2*20*4 instrs = 160 Cycles

40 Loop Iterations in 172 Cycles
10 Program Words

DO loops

- ✓ Executing single instruction or block of instructions
- ✓ Repeat times provided either using register (16-bit) or using immediate count loop (6-bit)
- ✓ Interruptible
- ✓ Nested hardware do-loops possible up to level 2
- ✓ Introduced instruction DOSLC, which assumes that the count has previously been download into LC register

REP loops

- ✓ Executing one single word instruction prescribed number of times
- ✓ Repeat times provided either using register (16-bit) or using immediate count loop (6-bit)
- ✓ Not interruptible

Interrupt Priority Structure

Interrupt Priority Level Summary

IPL	Description	Priority	Interrupt Sources
LP	Maskable	Lowest	SWILP Instruction
0	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #0 Instruction
1	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #1 Instruction
2	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #2 Instruction
3	Non-maskable	Highest	Illegal instruction, hardware stack overflow, SWI instruction, EOnCE Interrupts, misaligned data access

Interrupt accepted when IPL >= CCPL

Current Core Interrupt Priority Levels

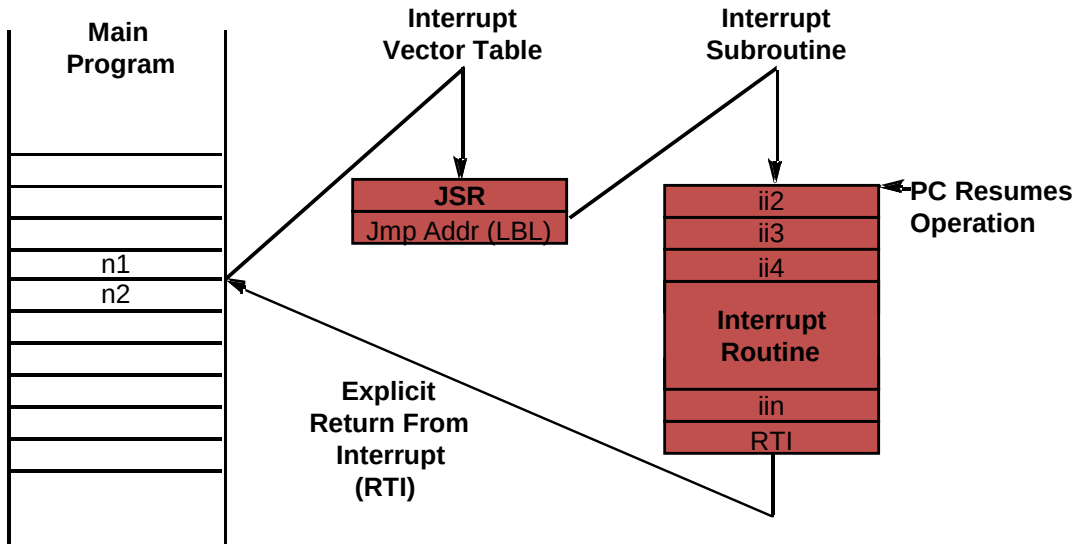
I1	I0	CCPL*	Exceptions Accepted	Exceptions Masked	Comments
0	0	0	IPL 0,1,2,3, and SWILP	None	All interrupts are allowed including the SWILP
0	1	1	IPL 1,2,3	IPL 0 and SWILP	All non-maskable interrupts and maskable interrupts that are programmed at level 1 or 2 are allowed
1	0	2	IPL 2,3	IPL 0 , 1 and SWILP	All non-maskable interrupts and maskable interrupts that are programmed at level 1 are allowed
1	1	3	IPL 3	IPL 0, 1, 2 and SWILP	Only non-maskable interrupts are allowed

* **CCPL:**Current Core Interrupt Priority Level

Standard Interrupt Processing

General Case:

- Vectored Interrupts - Vectors may be located anywhere in Program Memory (controlled by VBA)
- 4 Priority Levels - Highest is non-maskable
- Software Traps at each priority level
- One additional software trap (5th level) at lowest priority for O/S support

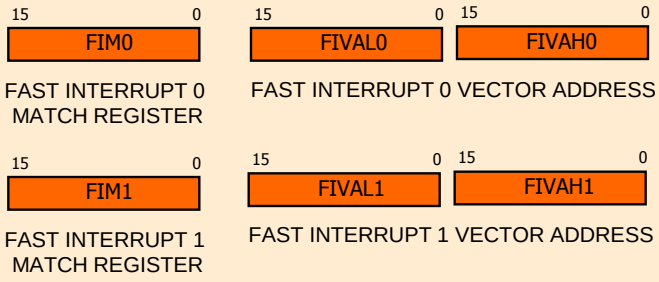


Fast Interrupts

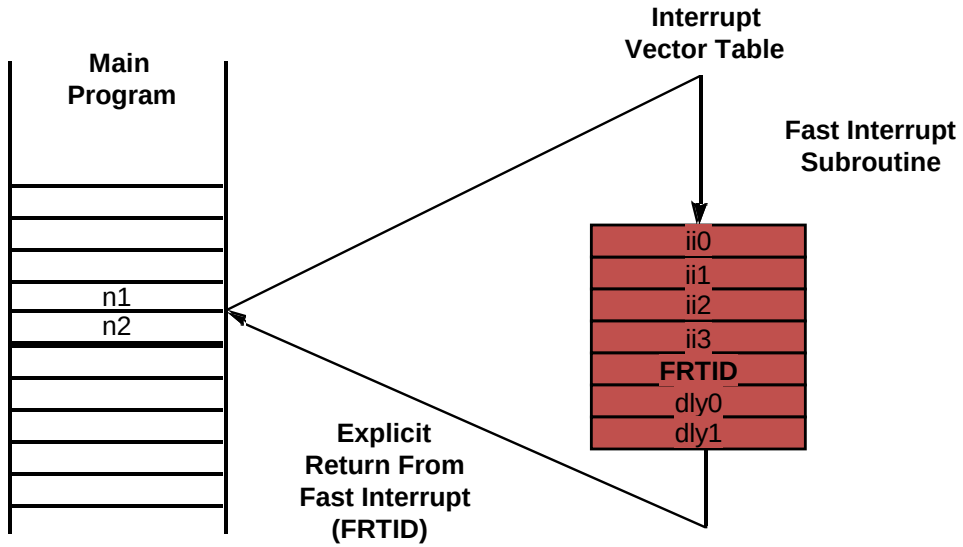
• Establishing of Fast Interrupts

- ✓ Setting the priority of the interrupt as level2, with the appropriate field in the IPR registers.
- ✓ Setting the FIM0 and FIM1 registers to the appropriate vector numbers.
- ✓ Setting the FIVAL0 and FIVAH0 registers with the address of the service code for the fast interrupt.
- ✓ The core then fetches the instruction from the indicated vector address and if it is not a JSR, the fast interrupt starts

ITCN FAST INTERRUPT REGISTERS



• Implemented Techniques



• Improved Latency and Throughput

- ✓ Overcome vectors and jumps directly to service routine
- ✓ Automatically swaps registers with shadows: R0, R1, N, and M01
- ✓ Automatically aligns SP and pushes the Y0 and Y1 registers onto the stack
- ✓ Automatically advances the SP to an empty 32-bit location and automatically restores above registers on exit

ADC Service Handling - Fast Interrupt (I.)

- **Description**

✓ Example demonstrates optimal service routine for servicing ADC interrupt.

✓ The ADC interrupt is generated at 80 kHz frequency; i.e. 5.3 us eight ADC conversions are processed.

✓ The timer TC2 is used to generate 80kHz square wave signal. This square wave signal is used to trigger ADC conversions.

✓ The ADC A block is set to execute 4 simultaneous conversions after each trigger.

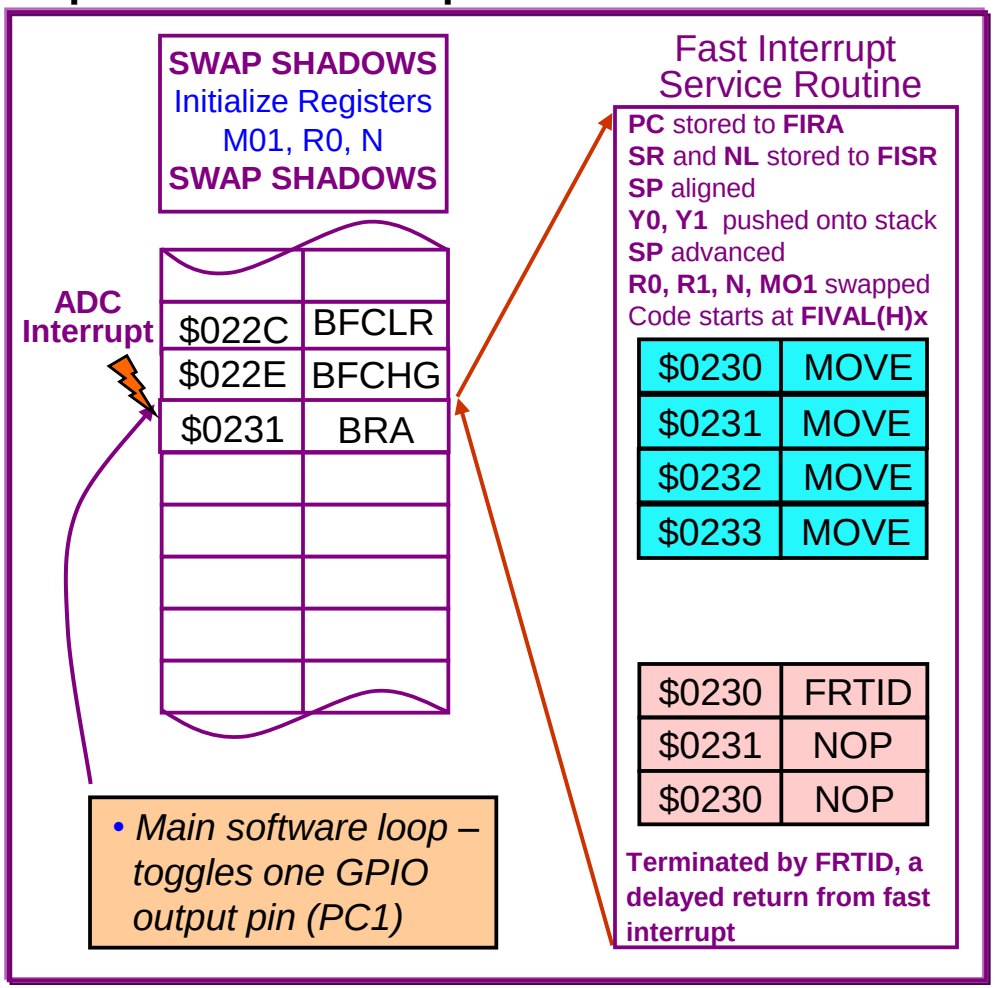
✓ Software continuously changes state of one GPIO pin in the background loop so the time of no pin change can be considered as Interrupt Execution Time.

- **Measured timings**

✓ Fast Interrupt – 0.84us

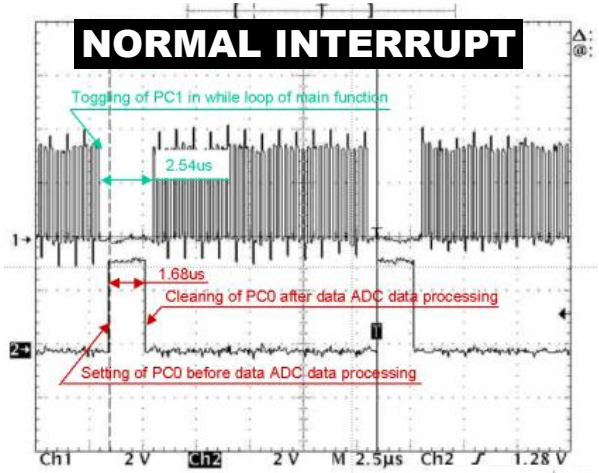
✓ Normal Interrupt – 2.54us

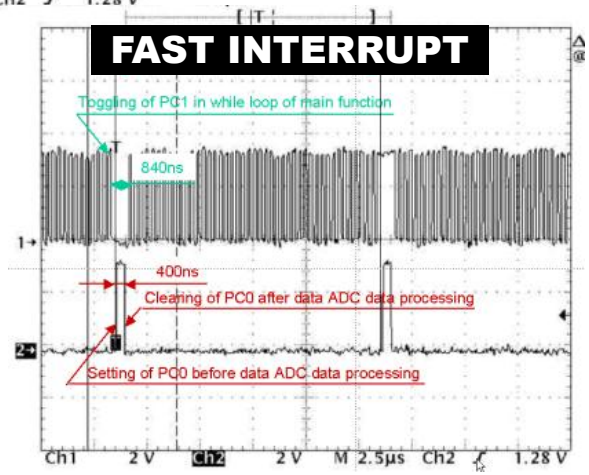
- **Implemented Techniques**



ADC Service Handling - Fast Interrupt (II.)

- **Interrupt Handling**

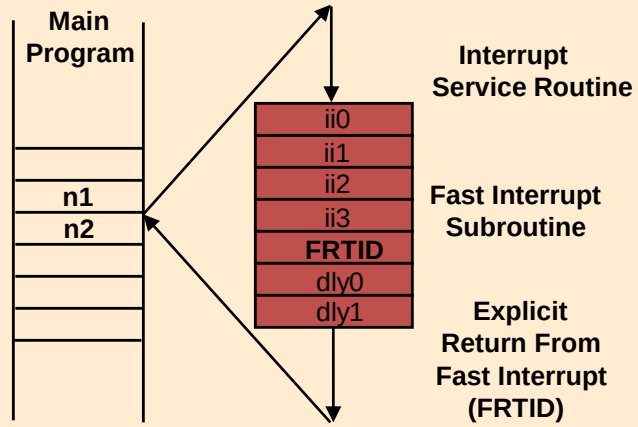




2-3x
faster execution then normal interrupt

- **Improved Latency and Throughput**

- ✓ Overcome vectors and jumps directly to service routine
- ✓ Operates at Interrupt Level 2 - Highest "maskable" priority
- ✓ Automatically swaps registers with shadows: R0, R1, N, and M01
- ✓ Automatically aligns SP and pushes the Y0 and Y1 registers onto the stack
- ✓ Automatically advances the SP to an empty 32-bit location and automatically restores above registers on exit

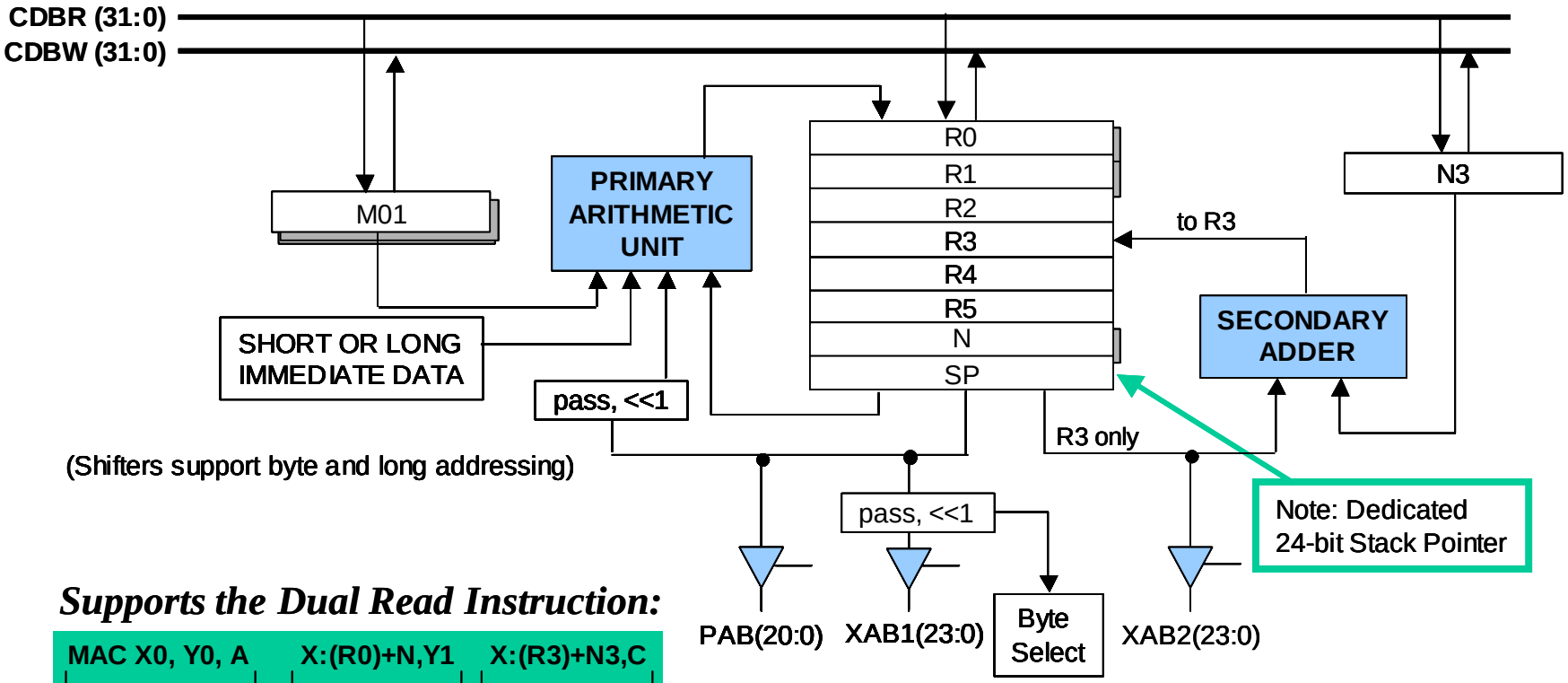


56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- **Address Generation Unit**
- Bit Manipulation Unit
- Instruction Pipeline
- Debugging Unit
- 56800E Core Highlights

AGU Block Diagram

- Provides up to 2 Data Memory Addresses per cycle
- Performs up to 2 Address Calculations after Issuing Addresses



Supports the Dual Read Instruction:

MAC X0, Y0, A	X:(R0)+N,Y1	X:(R3)+N3,C
Data ALU	Uses XAB1	Uses XAB2

Powerful Set of Addressing Modes

Indirect

- X:(Rn) No Update
- X:(Rn)+ Post Increment
- X:(Rn)- Post Decrement
- X:(Rn)+N Post Update by Register

Indexed

- X:(Rn+x) Indexed:3-bit Offset
- X:(SP-xx) Indexed:6-bit Offset
- X:(Rn+xxxx) Indexed:16-bit Offset
- X:(Rn+xxxxxx) Indexed:24-bit Offset
- X:(Rn+N) Indexed: By a Register

Supports 8, 16, 32-bits
Supports Modulo Arithmetic

Immediate

- #x 5-bit "Long" Constant
- #xx 6-bit Loop Ct
- #xx 7-bit Short
- #xxxx 16-bit
- #xxxxxxxx 32-bit

Absolute

- X:aa 6-bit Absolute Short
- X:<<pp 6-bit Peripheral Direct
- X:xxxx 16-bit Absolute
- X:xxxxxx 24-bit Absolute

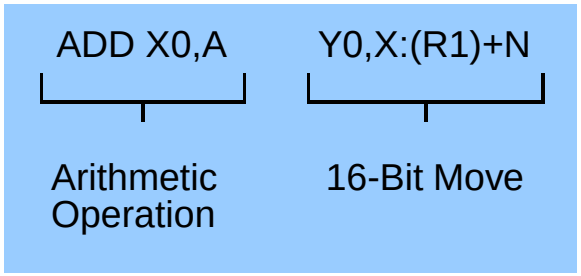
Other

- DDDDD Register Direct

56800E Parallel Move Instructions

Definition: One or two word sized moves occur in parallel with an arithmetic operation

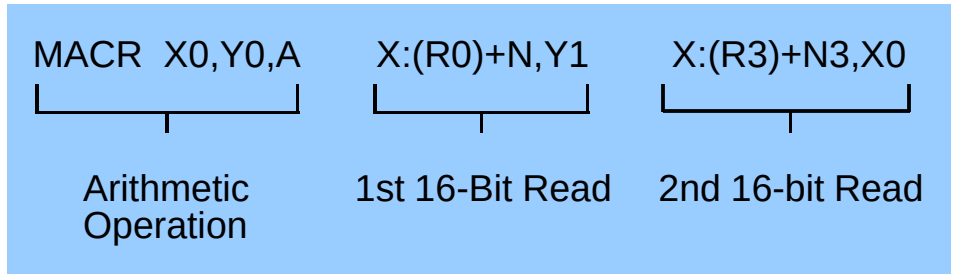
The "Single Parallel Move"



Examples:

MPYR	X0,Y0,A	X:(R0)+,X0
MAC	-X0,Y0,A	Y0,X:(R1)+N
ADD	A,B	Y1,X:(R2)+
TFR	Y1,A	A,X:(R3)+
INC.W	A	X:(R0)+,B
ASL	A	X:(R1)+N,C

The "Dual Parallel Read"



Examples:

MPY X0,Y0,A	X:(R0)+,Y0	X:(R3)+,X0
MACR X0,Y0,B	X:(R1)+,Y1	X:(R3)-,C
ADD X0,A	X:(R4)+N,Y0	X:(R3)+,X0
SUB Y1,B	X:(R4)+N,Y1	X:(R3)+N3,X0
MOVE.W	X:(R0)+N,Y0	X:(R3)+,C

Single Parallel Move Instructions

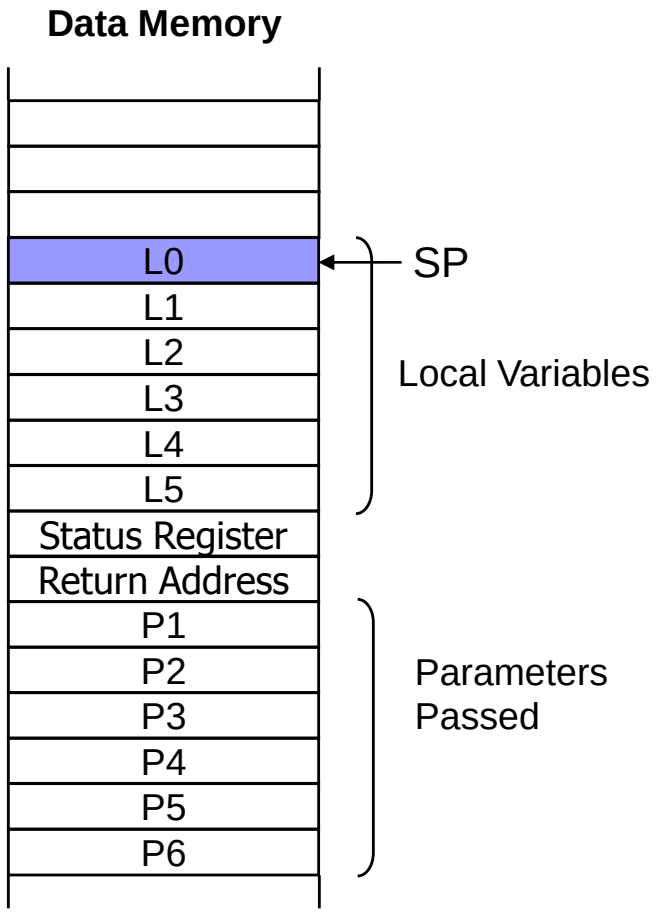
Data ALU Operation		Parallel Memory Move	
Operation	Operands	Source	Destination
MAC MPY MACR MPYR	Y1,X0,F Y0,X0,F Y1,Y0,F Y0,Y0,F A1,Y0,F B1,Y1,F	$X:(R_j)+$ $X:(R_j)+N$	X0 Y1 Y0 A B C A1 B1
MAC MPY MACR	C1,Y0,F C1,Y1,F		
MAC	–C1,Y0,F –C1,Y1,F	X0 Y1 Y0 A B C A1 B1	$X:(R_j)+$ $X:(R_j)+N$
ADD SUB CMP TFR	X0,F Y1,F Y0,F C,F A,B B,A		
SAT	F,Y0		
EOR.L	C,F		
ABS ASL ASR CLR RND TST INC.W DEC.W NEG	F		
SUBL[1]	A,D,B	X:(R1)+	AD

Dual Parallel Read Instructions

Data ALU Operation		First Memory Read		Second Memory Read	
Operation	Operands	Source 1	Destination1	Source 2	Destination 2
MAC MPY MACR MPYR	Y1,X0,F Y1,Y0,F Y0,X0,F C1,Y0,F	X:(R0)+ X:(R0)+N X:(R1)+ X:(R1)+N	Y0 Y1	X:(R3)+ X:(R3)–	X0
		X:(R4)+ X:(R4)+N	Y0	X:(R3)+ X:(R3)+N3	X0
ADD SUB	X0,F Y1,F Y0,F A,B B,A	X(R0)+ X(R0)+N X(R4)+ X(R4)+N	Y1	X(R3)+ X(R3)+N3	C
TFR	A,B B,A				
CLR ASL ASR	F				
MOVE.W					

Structured Programming - The Software Stack

- **Software Stack support for structured programming**
- **Supports Local Variables**
- **Supports Parameter Passing to a Function**
- **For both C and Assembly Code**
- **Utilizes strong set of SP Addressing Modes**



Example: Local Variable "L5" accessed as X:(SP-5)

Also Note:
JSR and interrupts automatically stack PC and SR

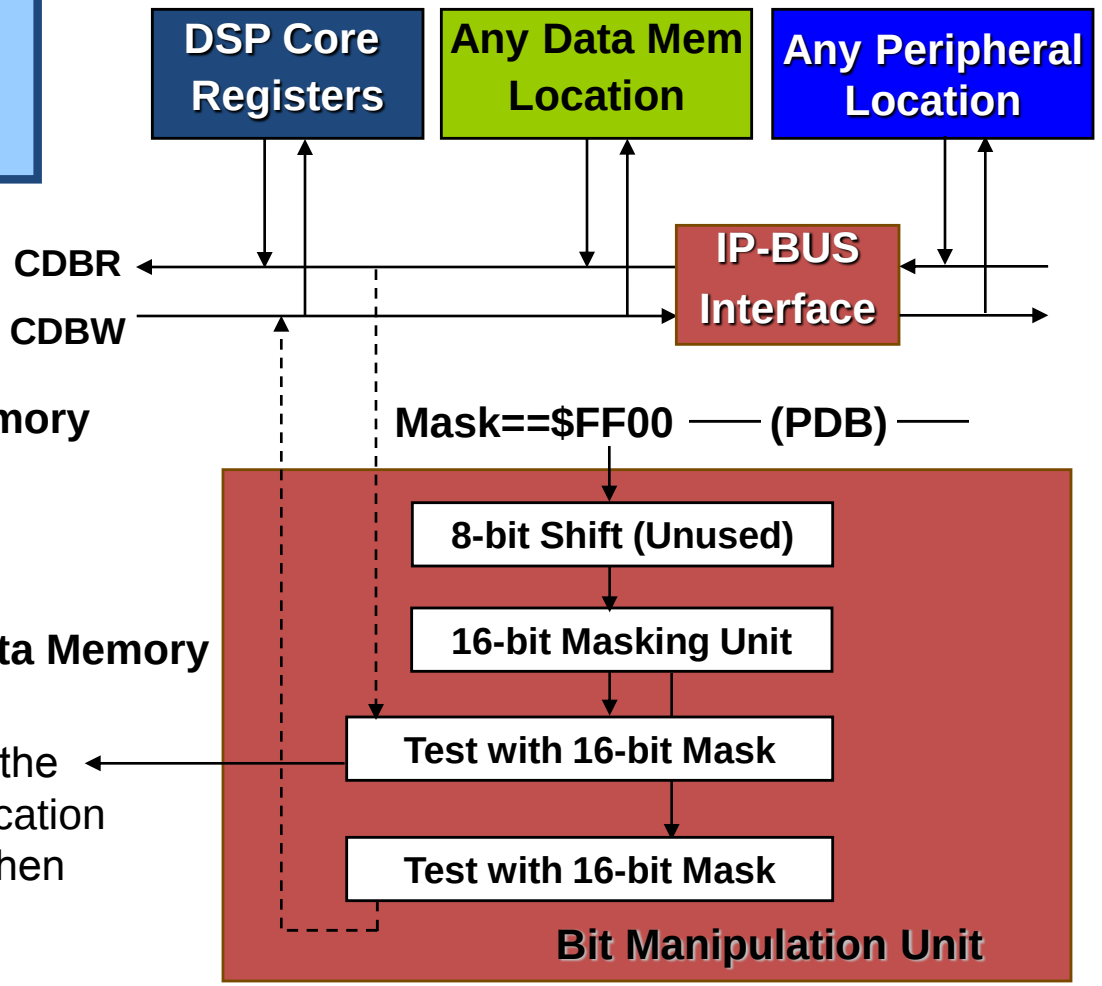
56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- Address Generation Unit
- **Bit Manipulation Unit**
- Instruction Pipeline
- Debugging Unit
- 56800E Core Highlights

Flexible Bit Manipulation Instructions

BFCLR	#$FF00	X:(R0)
Operation (clear bits)	16-Bit Mask	Operand in Memory

OPCODE: 8040 FF00



Steps

1. **Read 16-bit Word from Data Memory**
2. **Test Masked (upper 8) Bits**
3. **Clear Masked (upper 8) Bits**
4. **Write modified Word back to Data Memory**

Carry bit set to “1” if all bits in the Upper Byte of the Memory Location were all “1”’s; otherwise “0”. Then clears all selected bits.

Bit Manipulation Capabilities

Strong Set of Bit Manipulation Instructions:

- **BFSET**: Test and then set a field of bits in a word
- **BFCLR**: Test and then clear a field of bits in a word
- **BFCHG**: Test and then invert a field of bits in a word
- **BFTSTH**: Test a field of bits for all "1"s
- **BFTSTL**: Test a field of bits for all "0"s
- **BRSET**: Branch if a selected set of bits are all "1"s
- **BRCLR**: Branch if a selected set of bits are all "0"

Operates on any register or data memory location on the chip

Operates using a 16-bit mask

- Except: **BRSET** and **BRCLR** only allow an 8-bit mask on upper or lower byte

Other Bit Manipulation Instructions Performed Only in the Data ALU:

- 16-Bit Bidirectional Multi-bit Shifting (uses Data ALU registers)
- Arithmetic and Logical Shifts (uses Data ALU registers)
- Rotates (uses Data ALU registers)
- Increment and Decrement of Memory Locations

Example - BFSET Operation

- BFSET – Test Bitfield and Set
  - Test and then set a filed of bits in a word
  - This instruction performs a read-modify-write operation on the destination memory location or register
  - Requires two destination accesses
  - Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared
  - If all bits in the mask are cleared, the instruction executes two NOPs and sets the C bit

Example - BFSET Operation

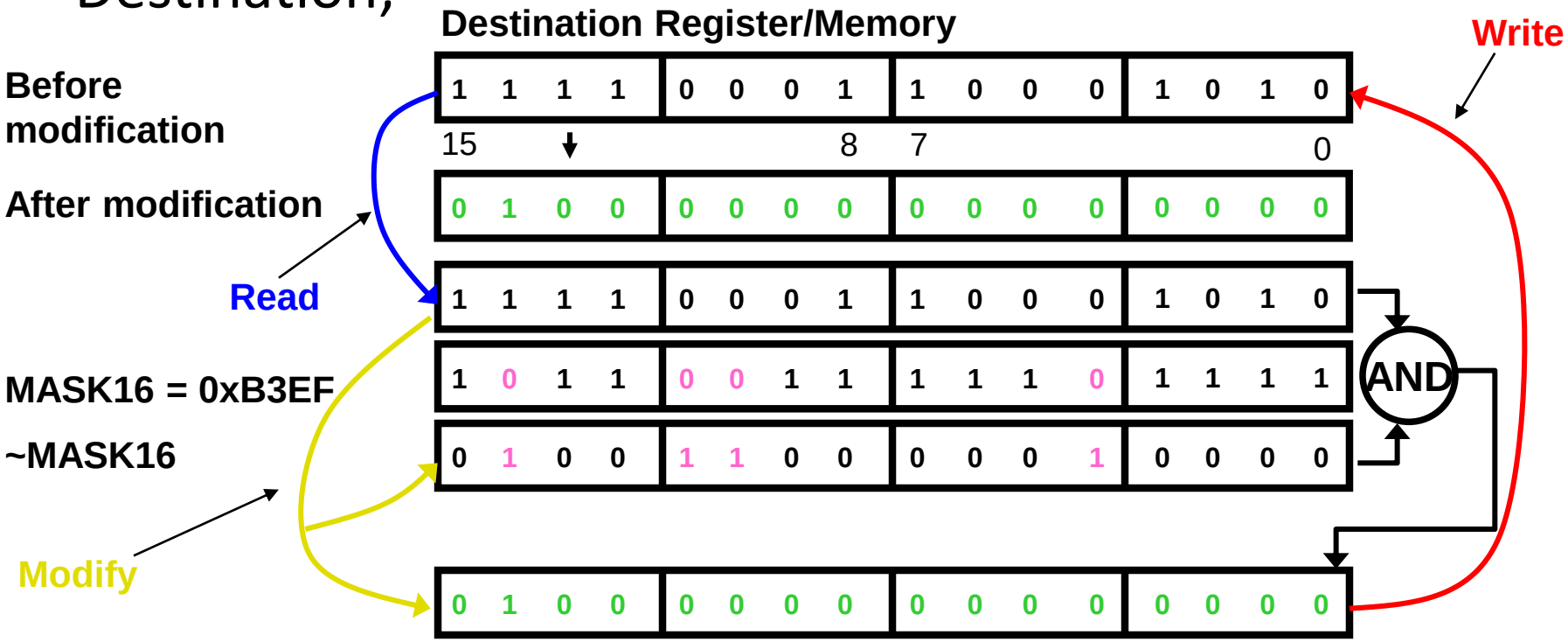
- BFSET #<MASK16>,Destination
- Operation: Destination = MASK16 | Destination;

Example - BFCLR Operation

- BFCLR – Test Bitfield and Clear
  - Test and then clear a filed of bits in a word
  - This instruction performs a read-modify-write operation on the destination memory location or register
  - Requires two destination accesses
  - Test all selected bits of the destination operand. If all selected bits are set, C is set; otherwise, C is cleared
  - If all bits in the mask are cleared, the instruction executes two NOPs and sets the C bit

Example - BFCLR Operation

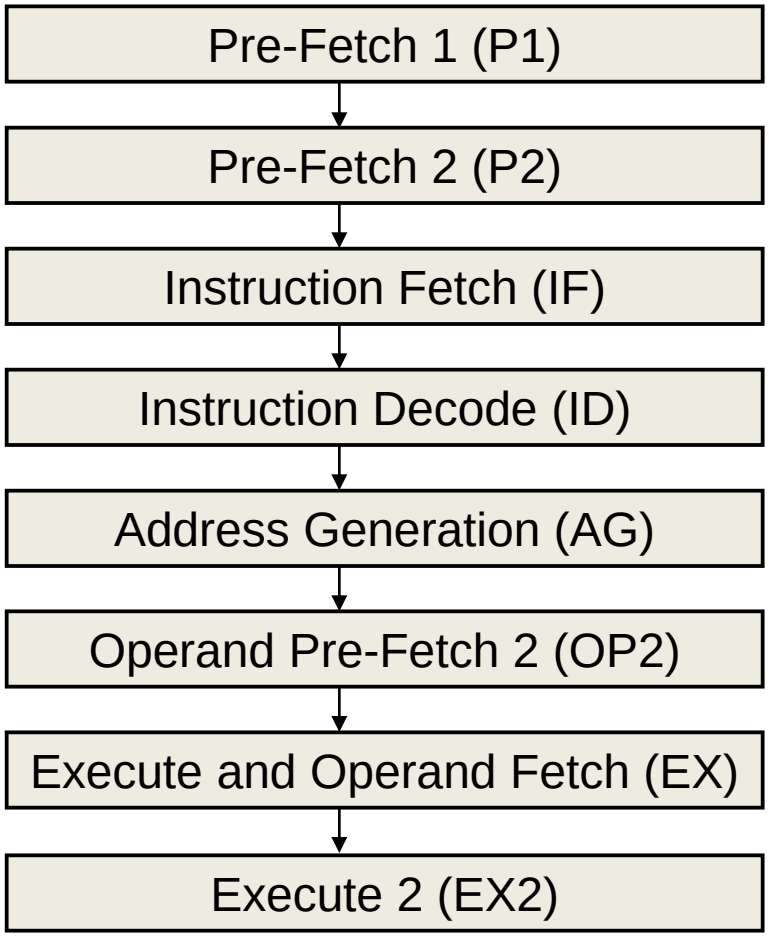
- BFCLR #<MASK16>,Destination
- Operation: Destination = ~MASK16 & Destination;



56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- Address Generation Unit
- Bit Manipulation Unit
- **Instruction Pipeline**
- Debugging Unit
- 56800E Core Highlights

Instruction Pipeline



- Eight-stage execution pipeline
- Higher execution throughput
- Instructions typically require 7 or 8 clock cycles to be fetched, to be decoded, and to finish execution
- Most instructions will complete and be retired by the end of the Execute stage
- AGU arithmetic instructions complete execution in the Address Generation stage

Instruction Pipeline

- MOVE.W X:(R0),A // n1: 1-word, 1-cycle instruction
- ADD A,B // n2: 1-word, 1-cycle instruction
- MOVE.W B,C // n3: 1-word, 1-cycle instruction
- MOVE.W C1,X:$0C00 // n4: 2-word, 2-cycle instruction
- INC.W C // n5: 1-word, 1-cycle instruction

Pipeline Stage	Instruction Cycle											
	1	2	3	4	5	6	7	8	9	10	11	•
P1 (Pre-Fetch 1)	n1	n2	n3	n4	n4e	n5	•	•	•	•	•	•
P2 (Pre-Fetch 2)		n1	n2	n3	n4	n4e	n5	•	•	•	•	•
IF (Instruction Fetch)			n1	n2	n3	n4	n4e	n5	•	•	•	•
ID (Instruction Decode)				mov[1]	add	mov	mov	mov	inc	•	•	•
AG (Address Generation)					mov	add	mov	mov	mov	inc	•	•
OP2 (Operand Pre-Fetch 2)						mov	add	mov	mov	mov	inc	•
EX (Execute and Operand Fetch)							mov	add	mov	mov	mov	•
EX2 (Execute 2)								—	—	—	—	—

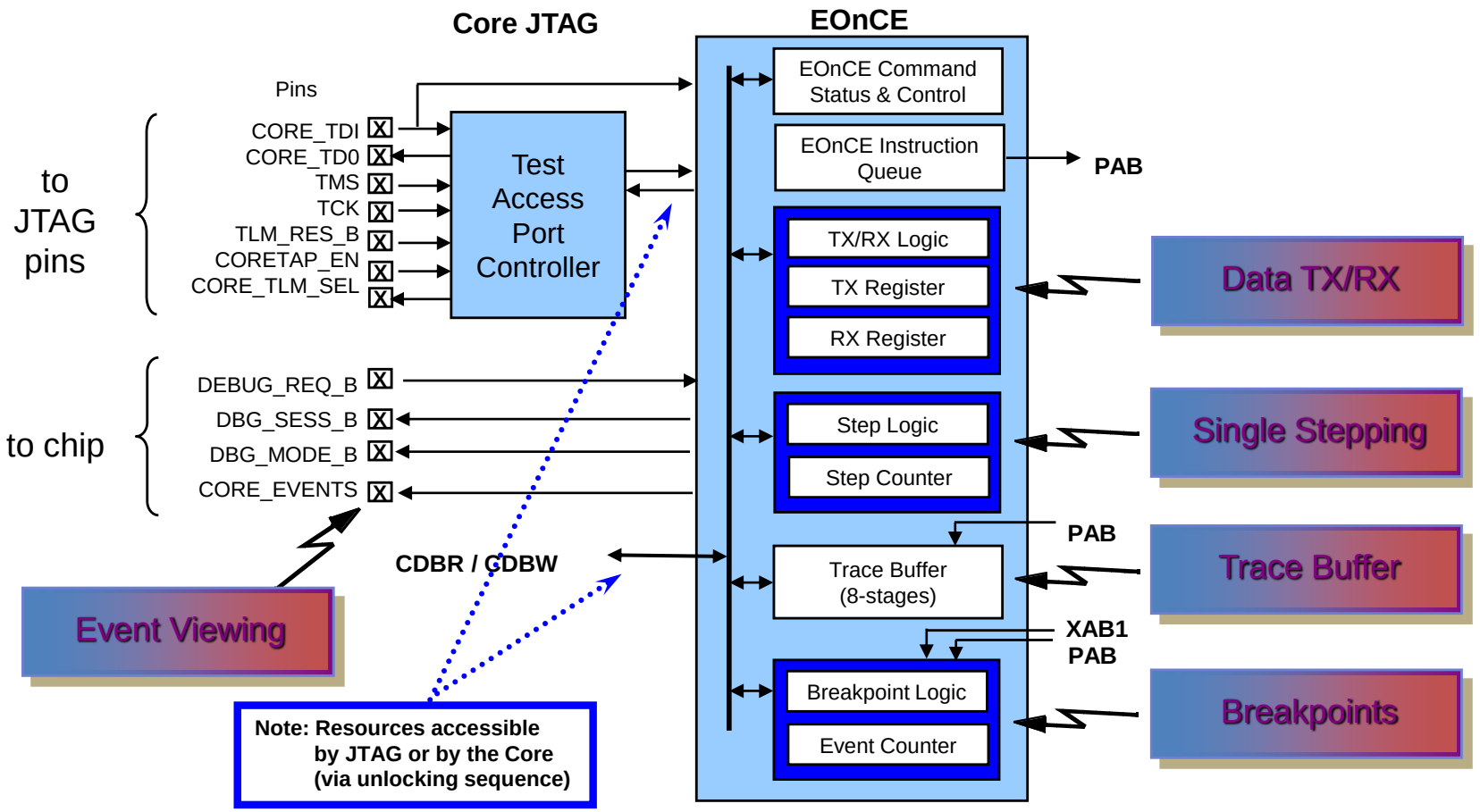
56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- Address Generation Unit
- Bit Manipulation Unit
- Instruction Pipeline
- **Debugging Unit**
- 56800E Core Highlights

Debugging - Realtime Emulation Capabilities

- System Level Debugging at one of 3 levels:
  - **Non-intrusive Realtime Debug**
  - **Minimally Intrusive Realtime Debug**
  - **Breakpoint and Step Mode — Core is halted**
- Nexus Level 0 Compliant
  - **A New Specification for on-chip Debugging Interface**
- Data Upload / Download through the JTAG port
- Advanced Breakpoint Capability
- Change of Flow Buffer
- Event Viewing through a terminal
- Resources accessible through JTAG or through the Core

Debugging - Core JTAG / EOnCE Interface



56800E Core

- Introduction
- Arithmetic Logic Unit
- Program Controller
- Address Generation Unit
- Bit Manipulation Unit
- Instruction Pipeline
- Debugging Unit
- **56800E Core Highlights**

56800E Core Highlights

- True stack pointer for C prog. efficiency
- 16-Bit program word for optimal code density
- General purpose register Files
- Orthogonal instructions available to the data and address register files
- 8, 16, and 32-bit data types
- Atomic Read-Modify-Write instructions
- Full set of bit manipulation instructions
- 16 and 32-bit shifting
- Nested Interrupt
- Fast interrupt support

56800E Core Highlights

- MAC instruction in one clock cycle
- Single and dual parallel move instructions
- No overhead hardware looping
- Nested looping capability
- Modulo arithmetic (for circular buffers)
- Fractional and integer arithmetic support
- Two types of saturation arithmetic
  - mode selectable
  - instruction based
- Accumulators with extension bits – 36-bit registers
- Shadow registers
- More addressing registers

Is it a 16 or 32-Bit MCU?

- Four 36-bit data registers
- Single cycle 8, 16, 32 bit data movement (inst.B, inst.W, inst.L)
- Eight 24-bit address registers
- 32-bit ALU operations
  - Add, Subtract, Test, Compare, Logical, etc.
  - Fractional 16 x 16 => 36-bit result 1 cycle
  - Fractional 16 x 32=> 36-bit result in 3 cycles
  - Frac/Signed based on 32 / 16 => 16-bit result in 16 cycles
- 32-bit shifting
- 24-bit pointers
- 24-bit pointer arithmetic

Thank you