Hardware Acceleration

Mark McDermott
Steven Smith

Agenda

- Taxonomy of Hardware Acceleration
- Microcoded Co-Processor
  - MC68332 Time Processing Unit
- ISA Enhancements:
  - HC12 Fuzzy Logic Acceleration
  - SIMD Instructions

Taxonomy of Hardware Acceleration

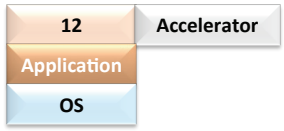
Common HW Acceleration Applications

- Graphics
- Data Compression
- Audio/Video Encoding/Decoding
- Image sensing and processing
- Data Encryption: RSA, DES, AES
- Router frame queuing, port selection

Hardware Acceleration Basics

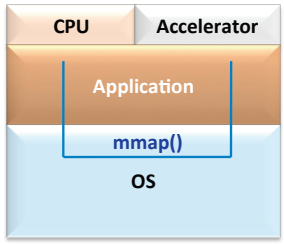
- Memory-mapped or directly connected to GP CPU
- Bus-based, FIFO, or register data interfaces
- Typically, the processor transfers data to the accelerator, issues a "go" command, and then collects result data later.
  - Polled or interrupt-based interface

Four Programmers Models of Accelerator Design

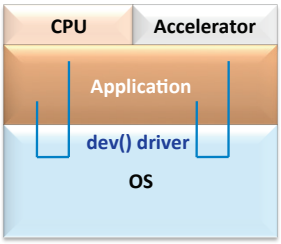


Base - HW I/F only

No OS Service (in simple embedded systems)



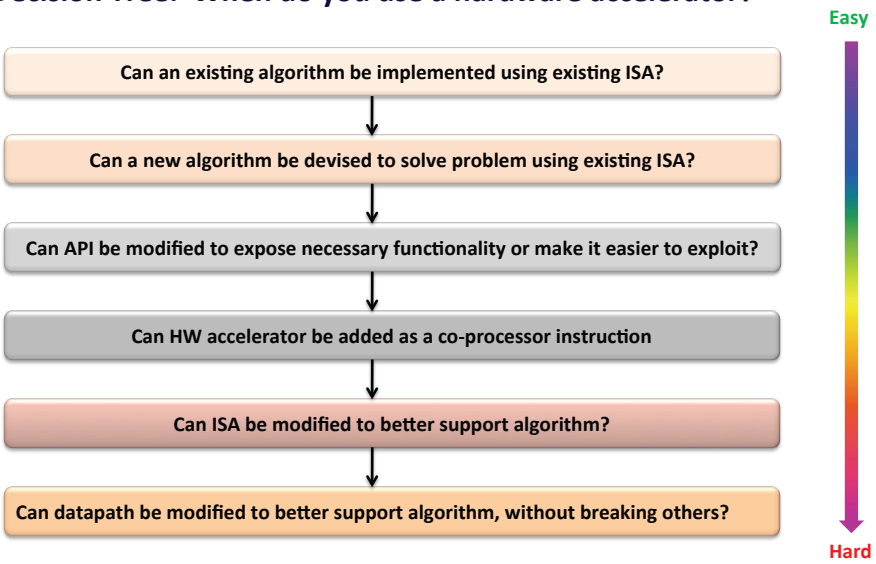
OS service – Accelerator accessed as a user space memory mapped I/O device



Virtualized Device with OS scheduling support

Perkowski, psu.edu

Decision Tree: When do you use a hardware accelerator?



Hardware Accelerator Interface: Polling

- **Polling interfaces usually require the processor to read a memory-mapped register to determine the state of the accelerator.**
  - Can the accelerator accept new input data?
  - Is the accelerator done with its current task?
  - Has the accelerator generated an error condition?

- **Polling interfaces offer minimal latency between the setting of a condition on the accelerator and its discovery by the controlling processor.**
  - But processor isn't doing other work while it polls…

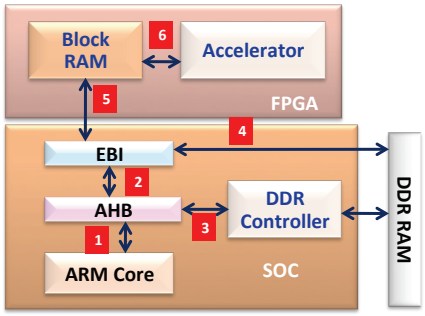
Hardware Accelerator Interface: Interrupts

- **Interrupt-based interfaces allow the accelerator to signal conditions to the controlling processor.**
  - Interrupt latency is longer than is achievable via the polling method.
  - But the processor can more easily proceed with other work while the accelerator is busy with a task.

- **Interrupts more efficient for coarse grained parallelism (i.e., larger tasks with looser and less frequent synchronization requirements)**

- **Interrupts may not work for real-time control tasks with tight schedules**

CPU-Accelerator Interface Example



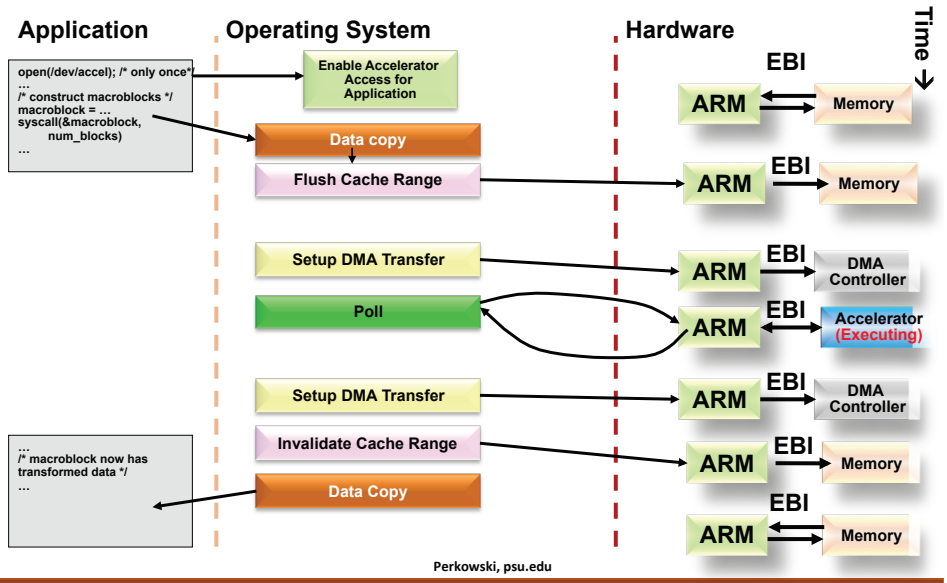
	Bus	First Access		Pipelined Access		Arbitration
		Read	Write	Read	Write	
1	ARM → AHB	2	2	2	2	
2	AHB → EBI	8	8	3	3	5
3	AHB → DDRC	4	4	4	4	
4	DDRC → DRAM	8	9	3	3	5
5	EBI → BRAM	20	20	8	8	12
6	BRAM → ACC	2	2	2	2	

- **EBI (External Bus Interface)**
  - 32 bit Bus
  - Access to DRAM data & FPGA data
  - 1/4 CPU frequency
  - Big penalty if bus is busy during first attempt to access bus

- **AHB (AMBA High Speed Bus)**
  - 32 bit bus
  - Runs at CPU clock frequency
  - Access to DDR Controller to provide addresses to SDRAM

Perkowski, psu.edu

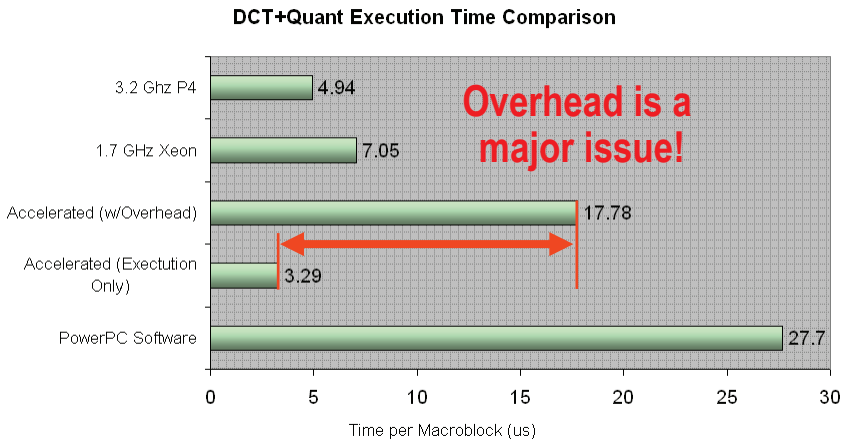
Typical CPU → Accelerator Transaction



Caching Issues with Accelerators

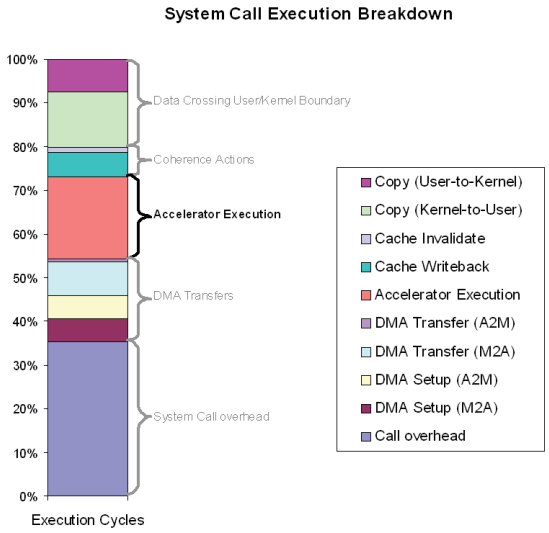
- Main memory provides the primary data transfer mechanism to the accelerator.
- Programs must ensure that caching does not invalidate main memory data.
  - CPU reads location S.
  - Accelerator writes location S.
  - CPU writes location S.
    - Program will not see proper value of S stored in the cache
- Many CPU buses implement test-and-set atomic operations that the accelerator can use to implement a semaphore. This can serve as a highly efficient means of synchronizing inter-process Communications (IPC)

Software Versus Hardware Acceleration



Perkowski, psu.edu

Device Driver Access Cost

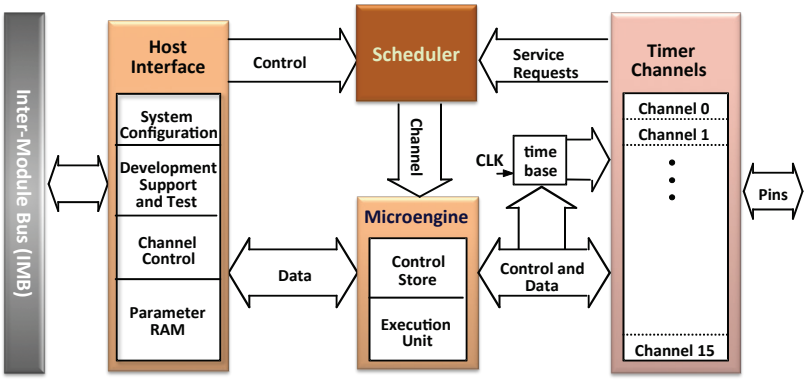


Perkowski, psu.edu

Micro-coded Co-Processor: MC 68332 Time Processing Unit

MC68332 Time Processing Unit

- The TPU3 can be viewed as a *special-purpose microcomputer* that performs a programmable series of two operations, match and capture.
- The microengine uses microcode to perform functions.



Time Processing Unit

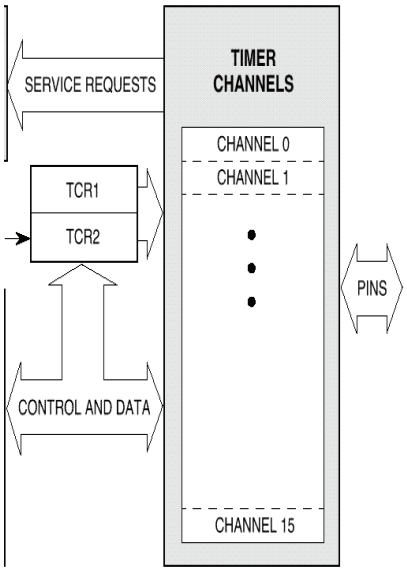
- Semi-autonomous microcontroller
- Operates concurrently with CPU
- Schedules tasks
- Processes ROM instructions
- Accesses shared data with CPU
- Performs Input/Output operations
- Programmable series of 2 operations
  - Match
  - Capture
- Each operation is called an ``event''
- A pre-programmed series of event is called a ``function''

TPU Preprogrammed Functions:

Function Number	Function Nickname	Function Name
0xF	PTA	Programmable Time Accumulator
0xE	QOM	Queued Output Match
0xD	TSM	Table Stepper Motor
0xC	FQM	Frequency Measurement
0xB	UART	Universal Asynchronous Receiver/Transmitter
0xA	NITC	New Input Capture/Input Transition Counter
9	COMM	Multiphase Motor Commutation
8	HALLD	Hall Effect Decode
7	MCPWM	Multi-Channel Pulse Width Modulation
6	FQD	Fast Quadrature Decode
5	PPWA	Period/Pulse Width Accumulator
4	OC	Output Compare
3	PWM	Pulse Width Modulation
2	DIO	Discrete Input/Output
1	SPWM	Synchronized Pulse Width Modulation
0	SIOP	Serial Input/output Port

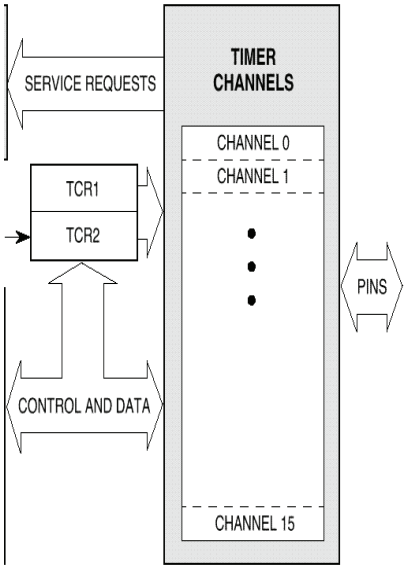
Time Bases

- Two sixteen-bit counters provide time bases for all
- Pre-scalers controlled by CPU via bit-fields in TPU module configuration register TPUCMR
- Current values accessible via TCR1 and TCR2 registers
- TCR1, TCR2 can be read/written by TPU microcode- not available to CPU
- TC1 qualified by system clock
- TC2 qualified by system clock or external clock



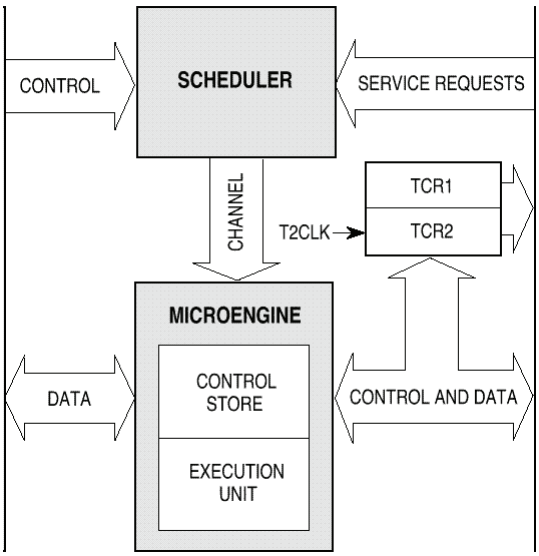
Timer Channels

- Sixteen channels
  - each one connect to a MCU pin
- Each channel has symmetric hardware:
- Event register
  - 16-bit capture register
  - 16-bit compare/match register
  - 16-bit comparator
- Pin control logic - pin direction determined by TPU microengine



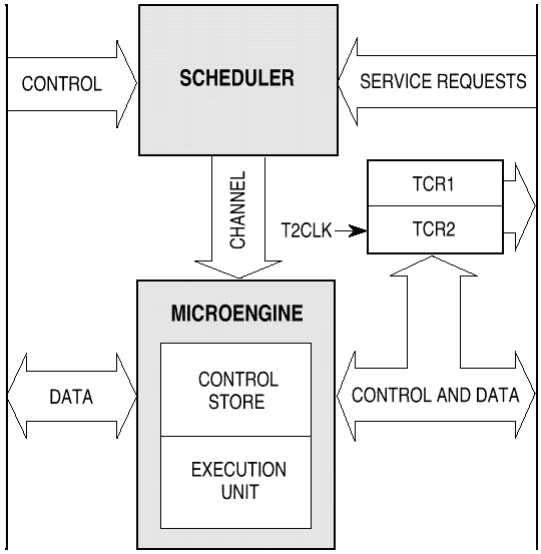
Scheduler

- Determines which of sixteen channels is serviced by the microengine
- Channel can request service for one of four reasons
  - host service
  - link to another channel
  - match event
  - capture event
- Host system assigns to each channel a priority
  - high
  - middle
  - low



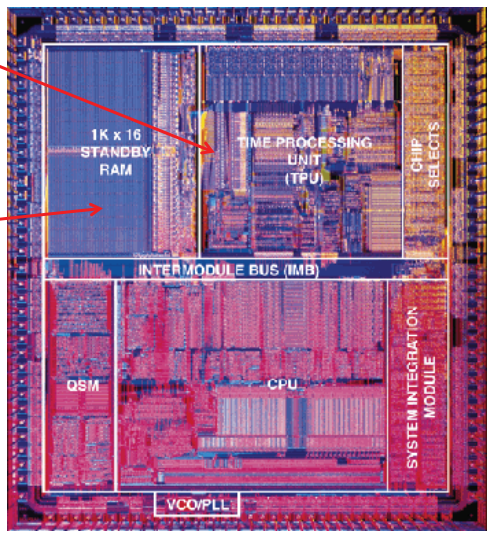
Microengine

- Executes microcoded functions for selected channel.
- Returns control to scheduler when completed.



WCS – Writeable Control Store

- The microcode in the TPU is hard coded into a mask programmable ROM.
- To facilitate microcode development and debug, a block RAM can be used to replace the ROM providing a "Writeable Control Store" capability.



68332 Die Photo

ISA Enhancements: Fuzzy Logic Acceleration

Overview

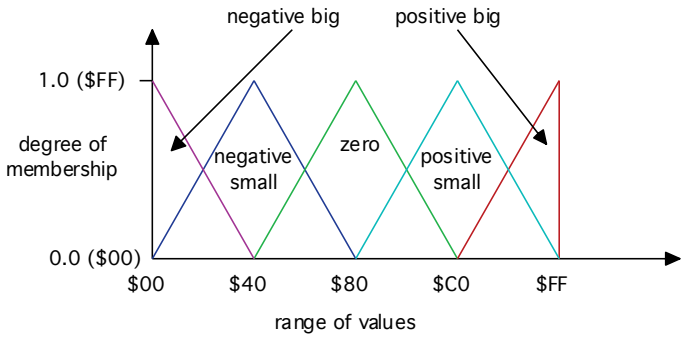
- **Fuzzy controllers provide a unique way of controlling complex systems.**
  - If written in software they can take a long time to execute, limiting their application in real-time systems.
  - Using dedicated logic and a few new assembler instructions, a microcontroller can be enhanced to execute a fuzzy controller quite efficiently.

Fuzzy Sets

- In the real world, very few things belong to a single classification and often the boundaries are not clear
- Fuzzy sets, then, is the extension of regular (Aristotelian) sets:
  - sets can overlap
  - members of a set have a degree of membership instead of just belonging to or not
- Fuzzy sets can be given linguistic labels such as warm, positive big, very cold, etc
- Fuzzy logic allows the sets to be computed
  - boundary conditions same as two-level logic

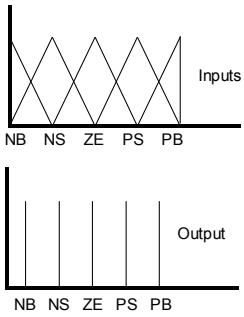
Input Memberships

- The degree to which an input belongs to a classification, is called its membership value
- The classifications can overlap so that an input value can belong to more than one, hence the ability to fuzzify the input



Motorola, Inc

Fuzzy Controller



Change of Error \ Error	NB	NS	ZE	PS	PB
NB	ZE	PS	PB	PB	PB
NS	NS	ZE	PS	PS	PB
ZE	NB	NS	ZE	PS	PB
PS	NB	NS	NS	ZE	PS
PB	NB	NB	NB	NS	ZE

Output Adjustment

- error = setting - position
- change of error = error - previous error
- output is an adjustment to current output

Example of "Fuzzy Rules"

- IF error is positive big THEN output is much smaller
- IF error is small and change of error is positive big THEN output is smaller
- IF error is small and change of error is positive small THEN output is a little smaller
- IF error and change of error are zero THEN output is zero

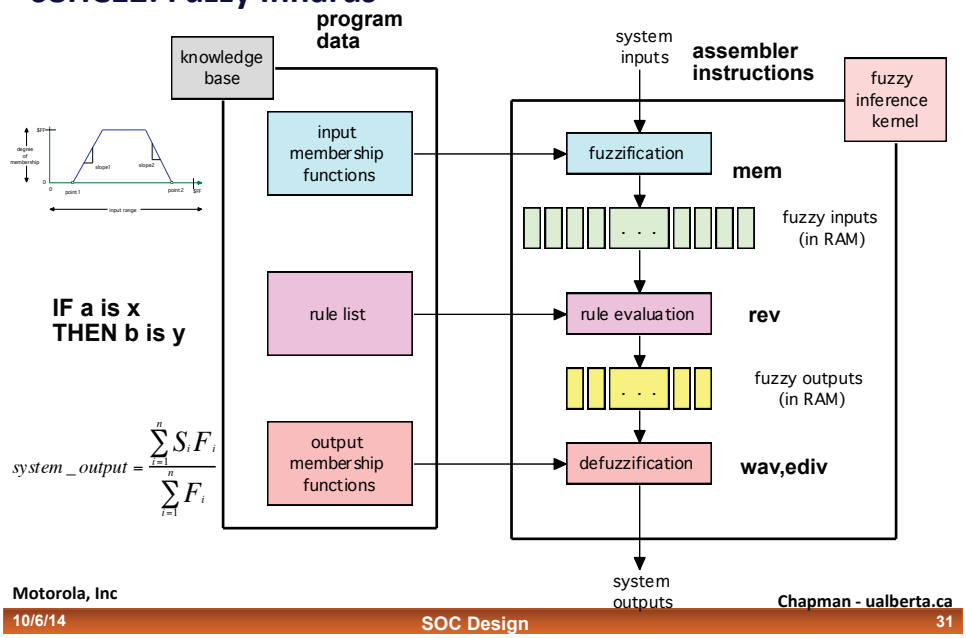
Fuzzy Logic Acceleration: 68HC11 vs. 68HC12

- 68HC12 microcontroller (Motorola)
  - Fuzzy logic instructions
- Simple fuzzy system response on 68HC11:
  - 750 ms
- Simple fuzzy system response on 68HC12:
  - 50 us (15,000 times faster)

68HC12 Fuzzy Instructions

- Native fuzzy instructions:
  - MEM; evaluate membership functions
  - REV; rule evaluation: IF a is x THEN b is y
  - WAV; weighted averaging
- Additional related instructions
  - MINA (place smaller of two unsigned 8-bit values in accumulator A)
  - EMIND (place smaller of two unsigned 16-bit values in accumulator D)
  - MAXM (place larger of two unsigned 8-bit values in memory)
  - EMAXM (place larger of two unsigned 16-bit values in memory)
  - TBL (table lookup and interpolate)
  - ETBL (extended table lookup and interpolate)
  - EMACS (extended multiply and accumulate signed 16-bit by16-bit to 32-bit)
- Orders of magnitude faster than fuzzy routines

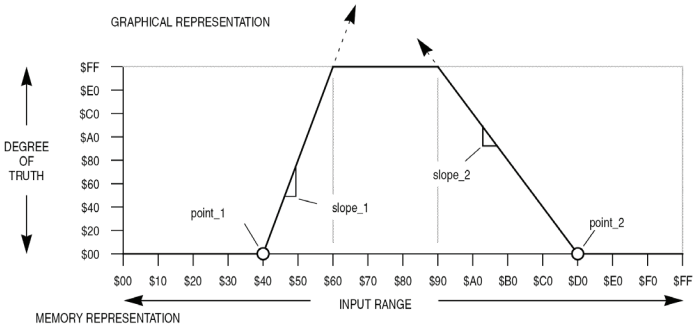
68HC12: Fuzzy Innards



A Closer Look

- **Each fuzzy instruction uses an efficient memory structure to maintain information.**
  - The MEM instruction uses an array of trapezoids for membership functions and writes to an array of bytes with the calculated membership values for each input.
  - The REV instruction use a byte array of offsets and flags for the rules antecedents and consequences and byte arrays for the outputs.
  - The REVW instruction uses an array of16 bit pointers and flags for antecedent and consequence values and a byte array for weights and outputs
  - The WAV instruction uses an byte array for outputs and singletons.

Trapezoidal Parameters



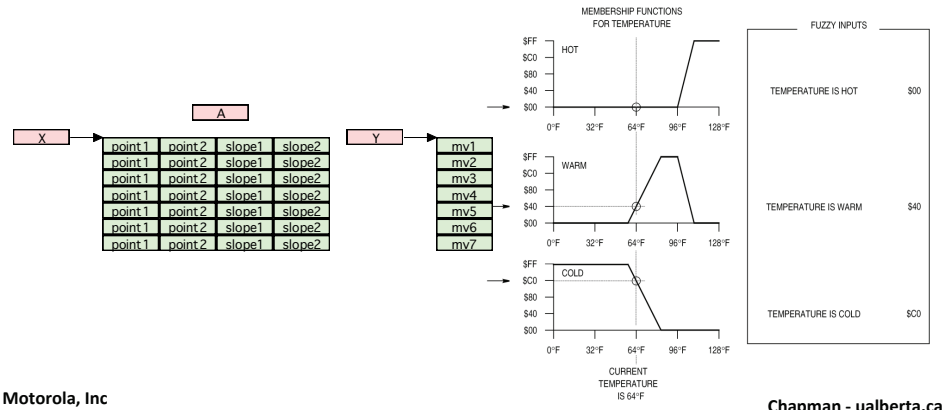
MEMORY REPRESENTATION

ADDR	$40	X-POSITION OF point_1
ADDR+1	$D0	X-POSITION OF point_2
ADDR+2	$08	slope_1 ($FF/(X-POS OF SATURATION – point_1))
ADDR+3	$04	slope_2 ($FF/(point_2 – X-POS OF SATURATION))

point1	point2	slope1	slope2
point1	point2	slope1	slope2
point1	point2	slope1	slope2
point1	point2	slope1	slope2
point1	point2	slope1	slope2
point1	point2	slope1	slope2
point1	point2	slope1	slope2

Fuzzify

```
fuzzify:  ldx #input_mfs   ; point at membership functions
          ldy   #fuz_ins    ; point at fuzzy input table
          ldaa  current_ins ; get first input values
          ldab  #7          ; 7 labels per input
loop:      mem              ; evaluate one membership func.
          dbne  b, loop     ; for 7 labels of one input
```

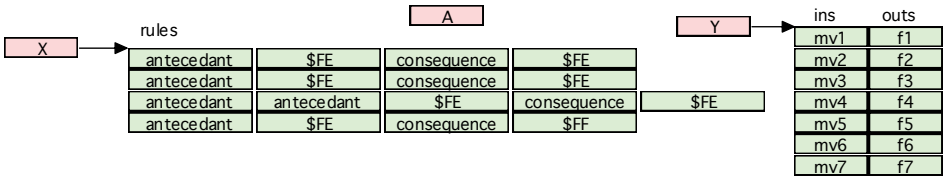


Rule Evaluation

- **Rule evaluation is the central element of a fuzzy logic inference program.**
  - Processes a list of rules from the knowledge base using current fuzzy input values from RAM to produce a list of fuzzy outputs in RAM.
- **The HC12 offers two variations of rule evaluation instructions:**
  - The REV instruction provides for unweighted rules (all rules are considered to be equally important).
  - The REVW instruction is similar but allows each rule to have a separate weighting factor which is stored in a separate parallel data structure in the knowledge base.
- **The fuzzy *and operator* corresponds to the mathematical minimum operation and the fuzzy *or* operation corresponds to the mathematical maximum operation.**

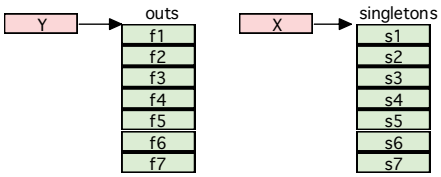
Evaluate Rules

```
        ldab #7           ; loop count
eval:   clr  1,y+         ; clear a fuzzy out and inc pointer
        dbne b, eval      ; loop to clr all fuzzy outs
        ldx  #rule_start  ; point at first rule element
        ldy  #fuz_ins     ; point at fuzzy ins and outs
        ldaa #$ff           ; init A (and clears V-bit)
        rev                 ; process rule list
```



De-Fuzzify

```
defuz: ldy  #fuz_out    ; point at fuzzy outputs
       ldx  #sgltn_pos  ; point at singleton positions
       ldab #7          ; 7 fuzzy outs per COG output
       wav              ; calculate sums for wtd av
       ediv             ; final divide for wtd av
       tfr  y,d         ; move result to A:B
       stab cog_out     ; store system output
```



$$system\_output = \frac{\sum_{i=1}^{n} S_i F_i}{\sum_{i=1}^{n} F_i}$$

Many different algorithms for defuzzification

$$Center\ of\ Gravity \qquad U = \left(\int_{min}^{max} u \times \mu(u)\ du\right) \div \int_{min}^{max} \mu(u)\ du \qquad (A.1)$$

$$Center\ of\ Gravity\ for\ Singletons \qquad U = \left(\sum_{i=1}^{n} u_i \times \mu_i\right) \div \sum_{i=1}^{n} \mu_i \qquad (A.2)$$

$$Left\ Most\ Maximum \qquad U = inf(u'), \mu(u') = sup\ (\mu(u)) \qquad (A.3)$$

$$Right\ Most\ Maximum \qquad U = sup(u'), \mu(u') = sup\ (\mu(u)) \qquad (A.4)$$

Variable	Meaning
U	Result of defuzzification
u	Output variable
N	Number of singletons
μ	Membership function after accumulation
i	Index
MIN	Lower limit of defuzzification
MAX	Upper limit of defuzzification
SUP	Largest value
INF	Smallest value

ISA Enhancements: SIMD Instructions

Media Instructions: MMX

- Multimedia applications tend to perform repetitive operations on large quantities of 8 and 16-bit data
  - Filtering
  - Compression
  - Rendering
- Intel's MMX technology is designed to speed-up multimedia and communications applications.
- The technology includes special instructions and data types that allow such applications to achieve a new level of performance.

MMX Introduction

- Processors enabled with MMX technology deliver enough performance to execute compute-intensive communications and multimedia tasks on the standard PC platform.

- Commonly accelerated applications include graphics, image processing, MPEG video, music synthesis, speech compression, speech recognition, games, video conferencing and more.

Key Attributes of MMX Target Applications

- Small integer data types
- Small, highly repetitive loops
- Frequent multiplies and accumulates
- Compute-intensive algorithms
- Highly parallel operations

UT ECE

MMX Highlights

- Single Instruction, Multiple Data (SIMD) technique
- 57 instructions beyond base x86 instruction set
- Eight 64-bit wide MMX registers
- Four new data types

UT ECE

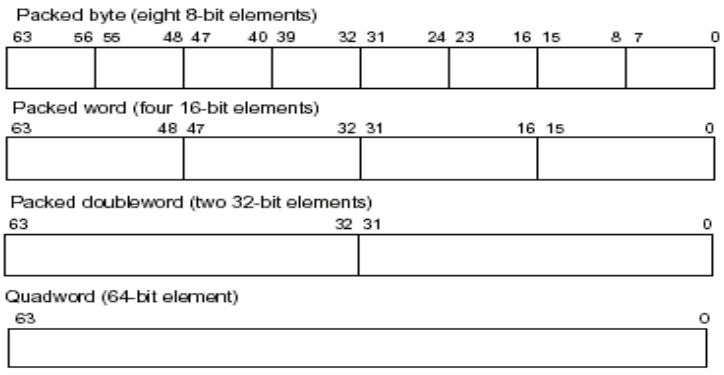
MMX SIMD

- Single Instruction, Multiple Data (SIMD)
- This allows many pieces of information to be processed with a single instruction, providing parallelism that greatly increases performance.
- Up to 8-way parallelism

MMX Data Types

- Packed Byte: Eight bytes packed into one 64-bit quantity
- Packed Word: Four 16-bit words packed into one 64-bit quantity
- Packed Double-word: Two 32-bit double words packed into one 64-bit quantity
- Quad-word: One 64-bit quantity

MMX Data Types in 64-bit Registers



MMX Instructions

- **The MMX instructions cover several functional areas:**
  - Basic arithmetic operations such as add, subtract, multiply, arithmetic shift and multiply-add
  - Comparison operations
  - Conversion instructions to convert between the new data types - pack data together, and unpack from small to larger data types
  - Logical operations such as AND, AND NOT,OR, and XOR
  - Shift operations
  - Data Transfer (MOV) instructions for MMX register-to-register transfers, or 64-bit and 32-bit load/store operations to memory

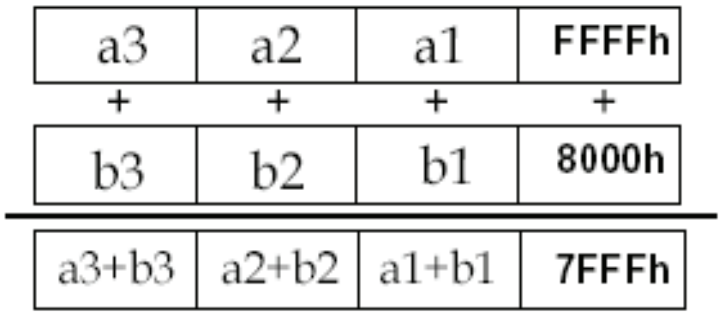
MMX Instruction Set Summary

Category	Mnemonic	Number of Different Opcodes	Description
Arithmetic	PADD[B,W,D]	3	Add with wrap-around on [byte, word, doubleword]
	PADDS[B,W]	2	Add signed with saturation on [byte, word]
	PADDUS[B,W]	2	Add unsigned with saturation on [byte, word]
	PSUB[B,W,D]	3	Subtraction with wrap-around on [byte, word, doubleword]
	PSUBS[B,W]	2	Subtract signed with saturation on [byte, word]
	PSUBUS[B,W]	2	Subtract unsigned with saturation on [byte, word]
	PMULHW	1	Packed multiply high on words
	PMULLW	1	Packed multiply low on words
	PMADDWD	1	Packed multiply on words and add resulting pairs
Comparison	PCMPEQ[B,W,D]	3	Packed compare for equality [byte, word, doubleword]
	PCMPGT[B,W,D]	3	Packed compare greater than [byte, word, doubleword]
Conversion	PACKUSWB	1	Pack words into bytes (unsigned with saturation)
	PACKSS[WB,DW]	2	Pack [words into bytes, doublewords into words] (signed with saturation)
	PUNPCKH [BW,WD,DQ]	3	Unpack (interleave) high-order [bytes, words, doublewords] from MMX™ register
	PUNPCKL [BW,WD,DQ]	3	Unpack (interleave) low-order [bytes, words, doublewords] from MMX register

MMX Instruction Set Summary (2)

Logical	PAND	1	Bitwise AND
	PANDN	1	Bitwise AND NOT
	POR	1	Bitwise OR
	PXOR	1	Bitwise XOR
Shift	PSLL[W,D,Q]	6	Packed shift left logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value
	PSRL[W,D,Q]	6	Packed shift right logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value
	PSRA[W,D]	4	Packed shift right arithmetic [word, doubleword] by amount specified in MMX register or by immediate value
Data Transfer	MOV[D,Q]	4	Move [doubleword, quadword] to MMX register or from MMX register
FP & MMX State Mgmt	EMMS	1	Empty MMX state

PADDW Instruction



PADD[W]: Wrap-around Add

PADDSUW Instruction

a3	a2	a1	FFFFh
+	+	+	+
b3	b2	b1	8000h
a3+b3	a2+b2	a1+b1	FFFFh

PADDUS[W]: Saturating Arithmetic

PMADDWD Instruction

a3	a2	a1	a0
*	*	*	*
b3	b2	b1	b0
a3*b3+a2*b2		a1*b1+a0*b0	

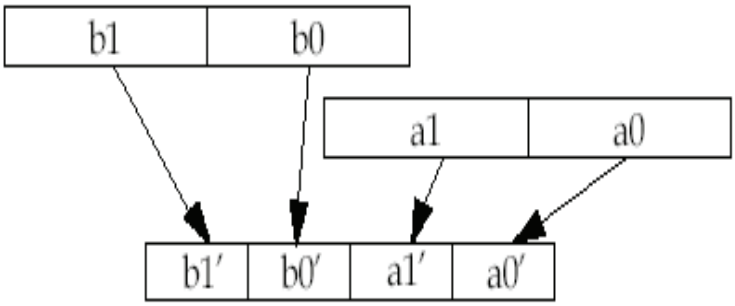
PMADDWD: 16b x 16b -> 32b Multiply Add

PCMPGTW Instruction

23	45	16	34
gt ?	gt ?	gt ?	gt ?
31	7	16	67
0000h	FFFFh	0000h	**0000h**

PCMPGT[W]: Parallel Compares

PACKSS[DW] Instruction



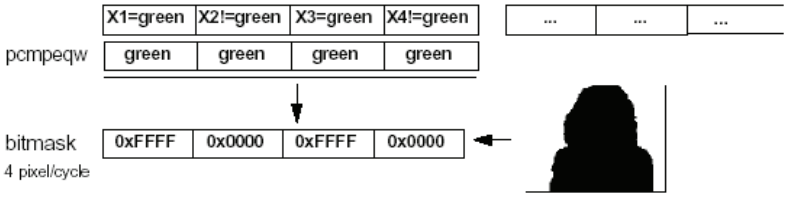
PACKSS[DW]: Pack Instruction

MMX Applications

Chroma Keying

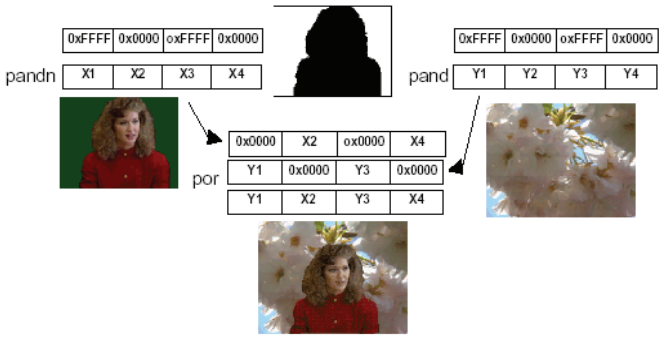
MMX Applications: Chroma Keying

pcmpeqw



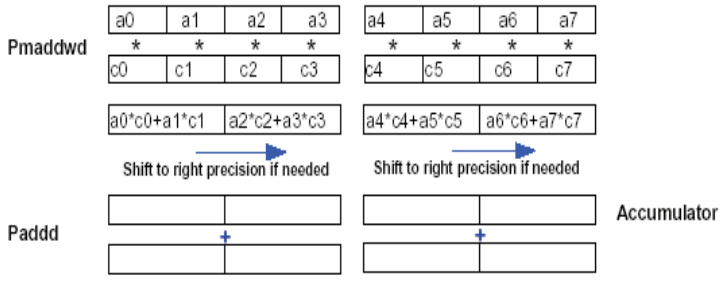
MMX Applications: Chroma Keying

pandn



MMX Applications: Vector Dot product

$$x = \sum a(i) * c(i)$$

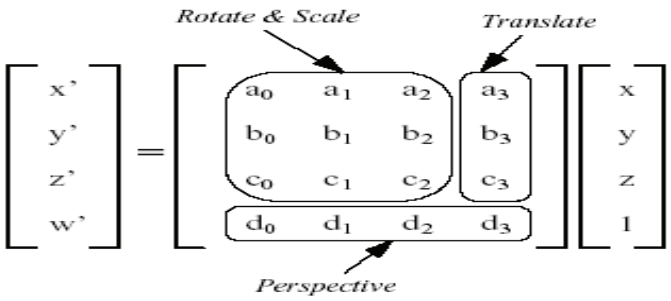


MMX Applications: Vector Dot product

Comparing instruction counts with and without MMX technology for this operation yields the following:

	Number of Instructions without MMX™ Technology	Number of MMX Instructions
Load	16	4
Multiply	8	2
Shift	8	2
Add	7	1
Miscellaneous	-	3
Store	1	1
Total	40	13

MMX Applications: Matrix Multiplication



$$x' = a_0x + a_1y + a_2z + a_3$$

MMX Applications: Matrix Multiplication

Instruction counts

	Number of Instructions without MMX™ Technology	Number of MMX instructions
Load	32	6
Multiply	16	4
Add	12	2
Miscellaneous	8	12
Store	4	4
Total	72	28