Efektivní programování mikropočítačů v embedded aplikacích

Ing. Jaroslav Lepka
Ing. Pavel Grasblum, Ph.D.

29. – 30. listopadu 2012

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

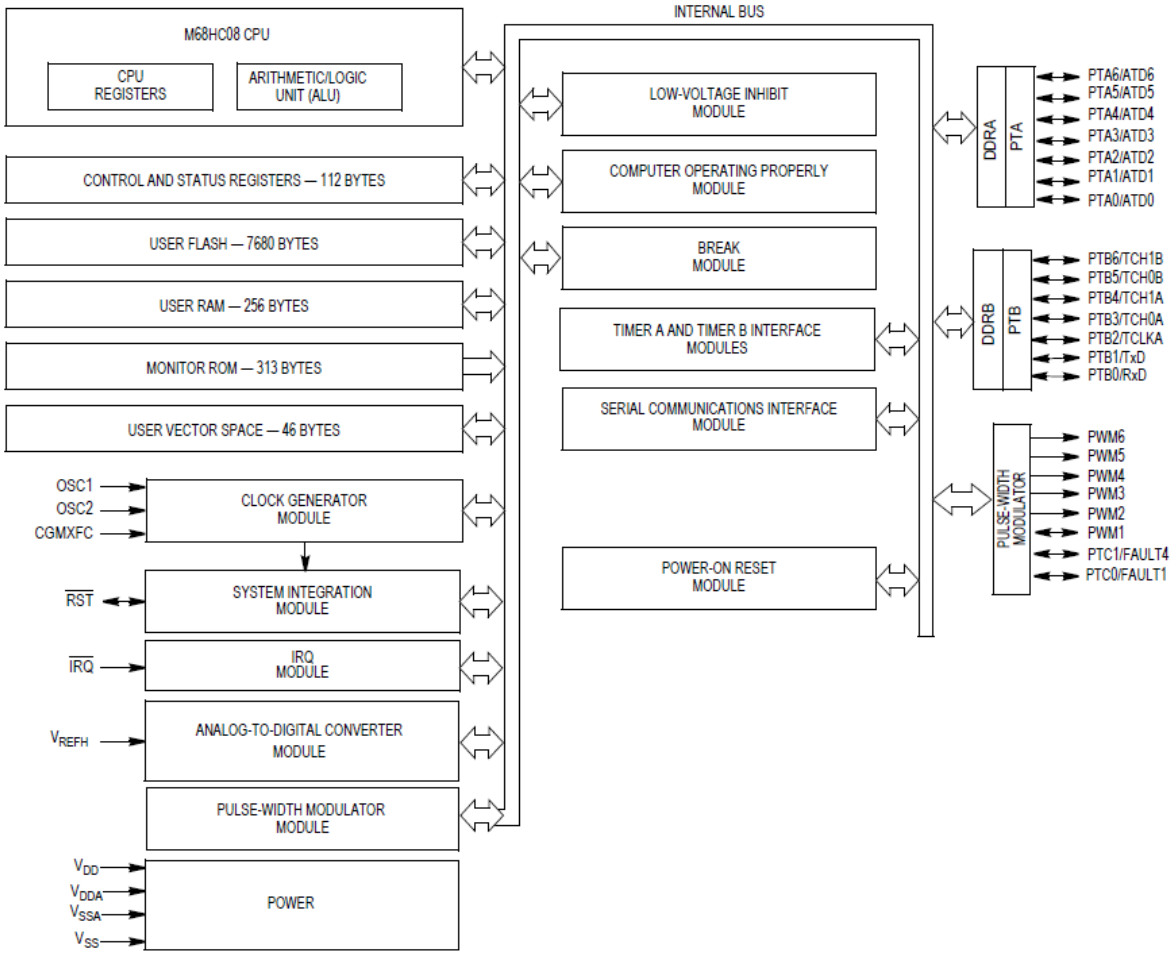
Časový rozvrh

- 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

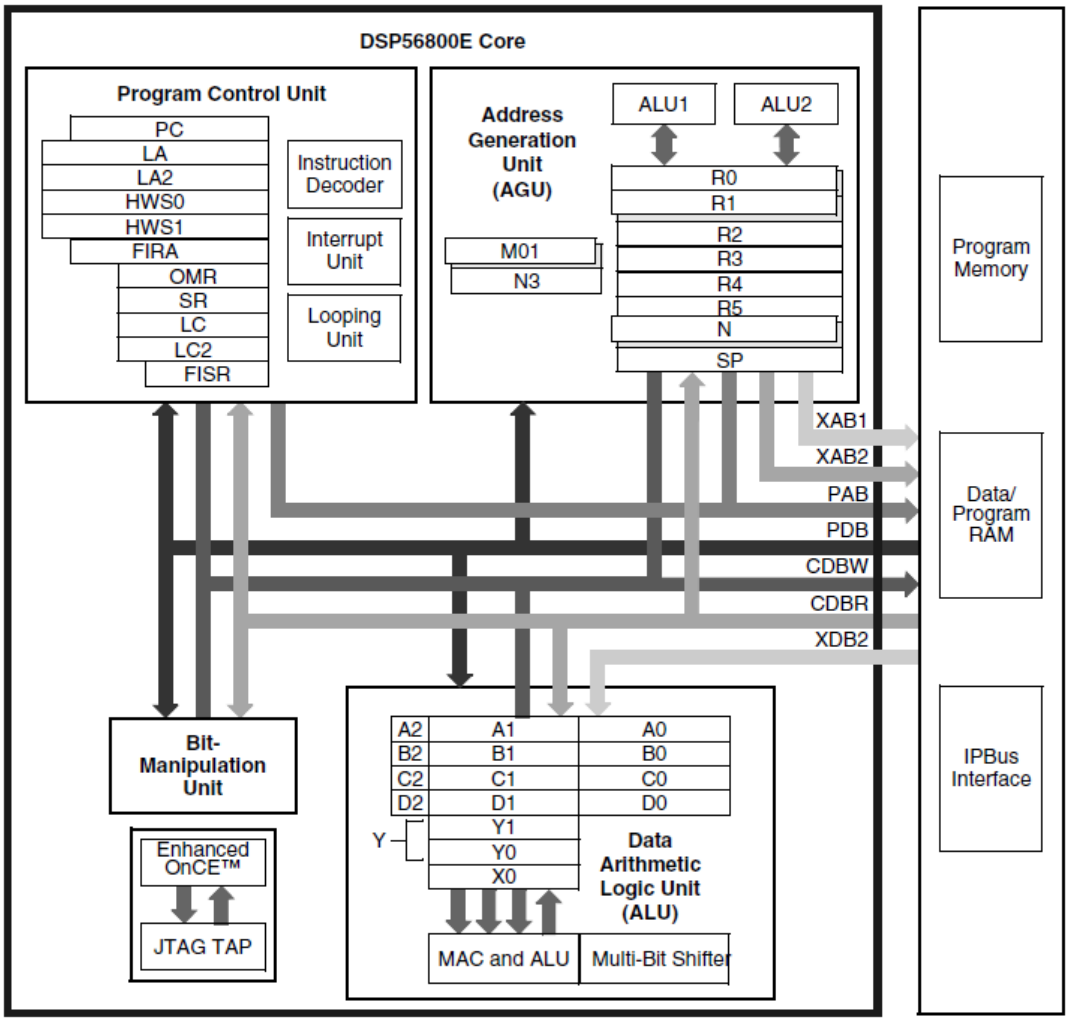
Understanding MCU Performance

- Old fashioned or simple MCUs have
  - Single master bus interface
    - All MCU busses are controlled by MCU core only. The MCU Core doesn't need to share the busses with another master

MC68HC908MR8 Bus Structure



DSC 56F800E Bus Structure



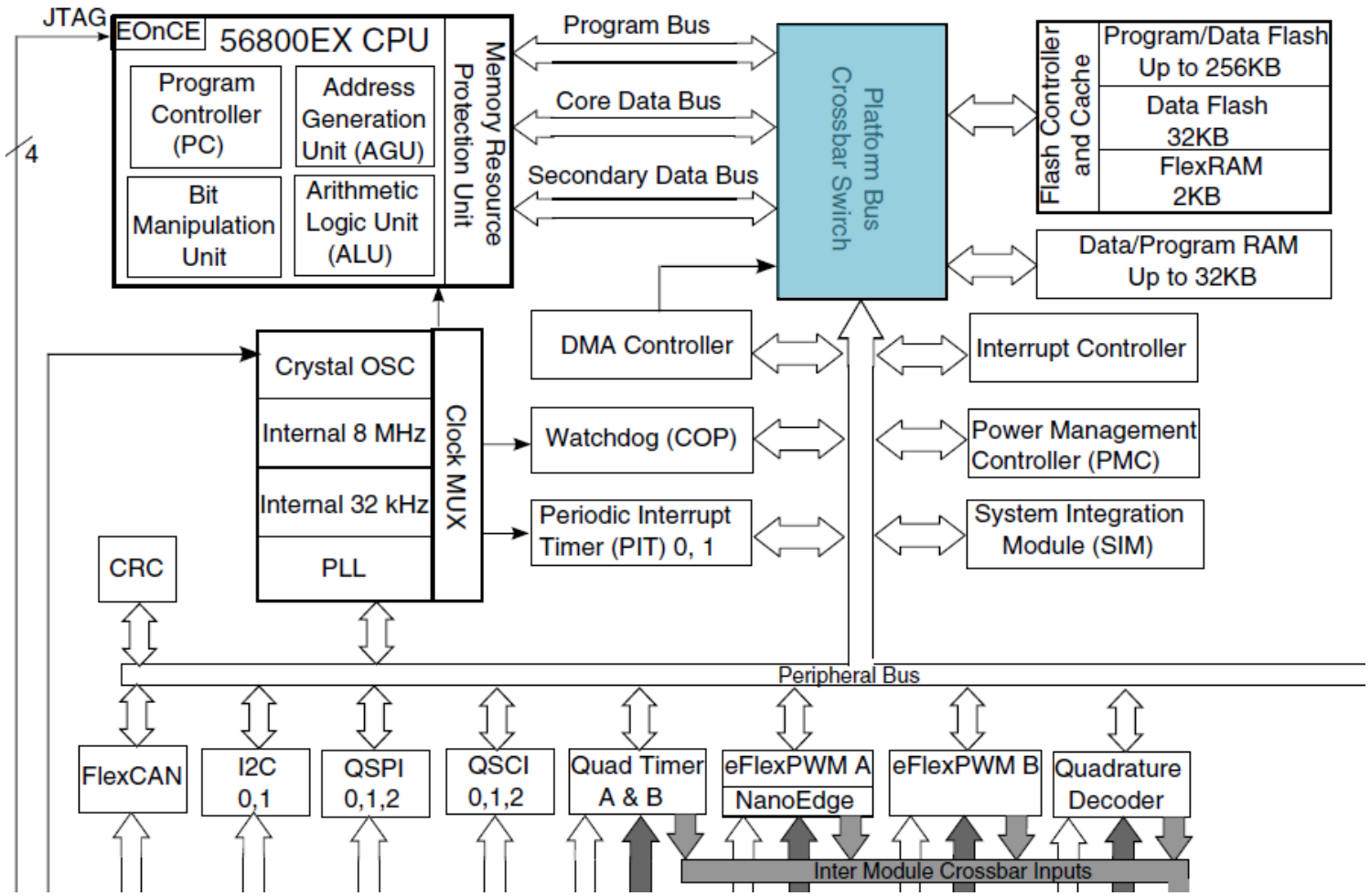
Understanding MCU Performance

- Old fashioned or simple MCUs have
  - Single master bus interface
    - All MCU busses are controlled by MCU core only. The MCU Core doesn't need to share the busses with another master
  - Core clock frequency <= Flash memory clock
    - The program code can be executed from the flash memory in the full speed without wait states
    - The current flash technology is quite slow compare to MCU core clock (~30 MHz)

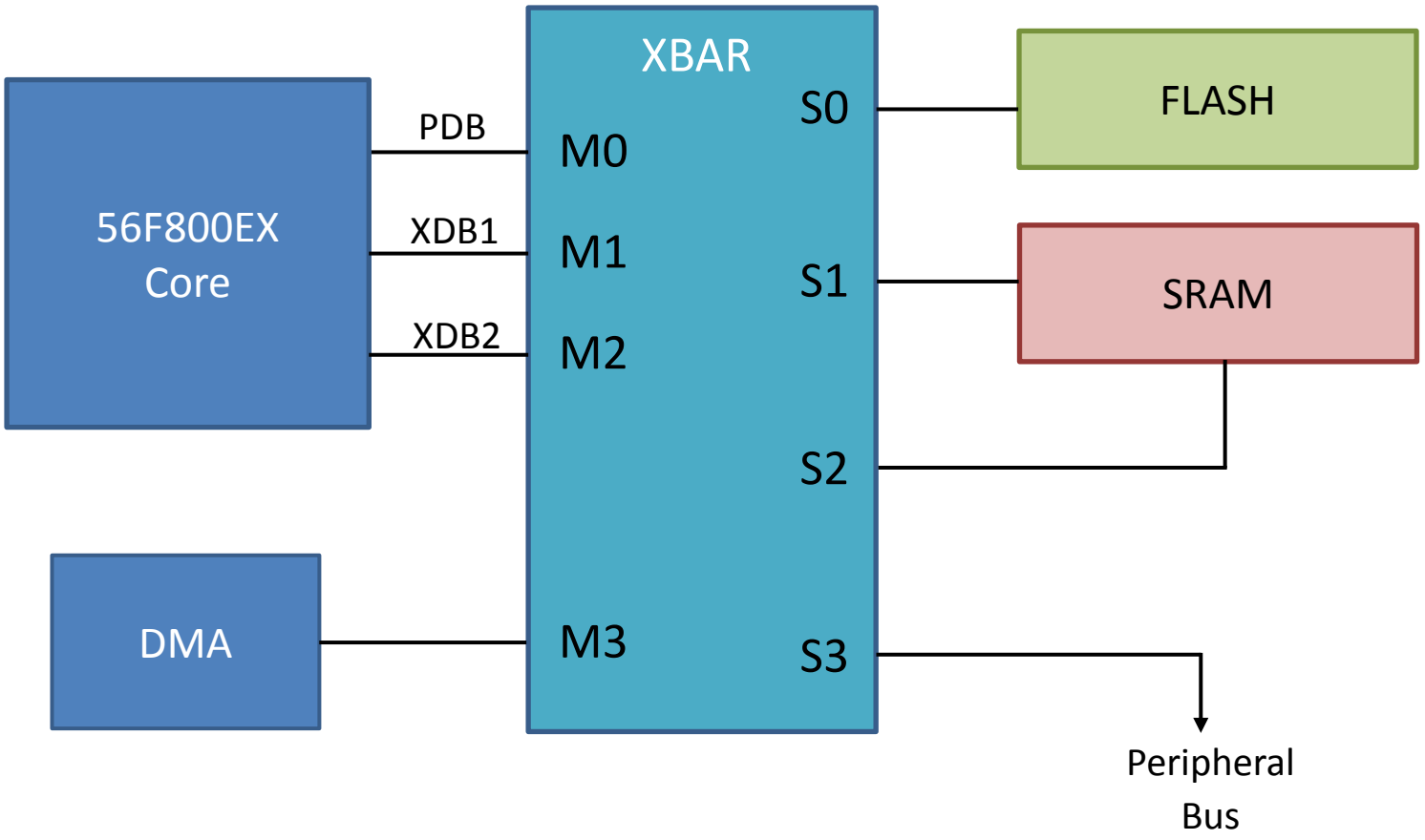
Understanding MCU Performance

- Modern powerful MCUs have
  - Multiple masters bus interface
    - There are more masters which share busses and peripherals
  - Core clock frequency >> Flash memory clock
    - The core frequency is much higher than maximal flash memory clock (>>~30 MHz)

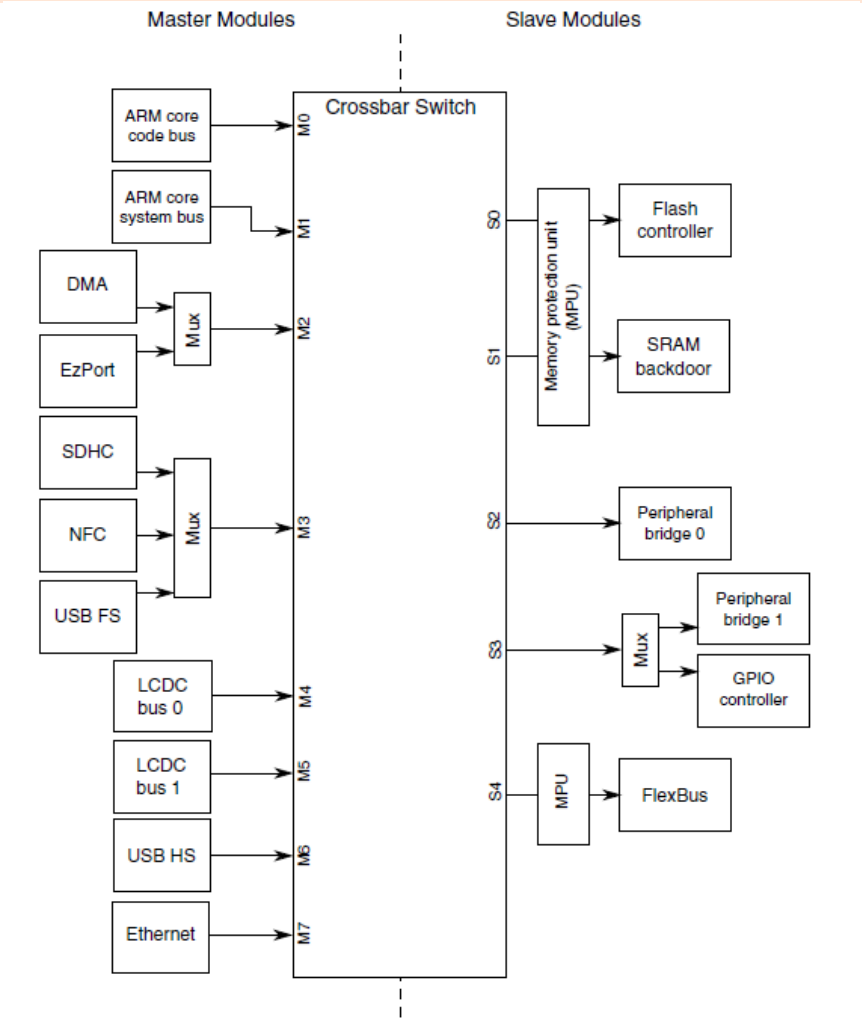
56F84xx Bus Structure



56F84xx Bus Structure Detail



Kinetis K70 Bus Structure



Crossbar Switch (AXBS)

- The crossbar switch connects bus masters and bus slaves using a crossbar switch structure
- This structure allows all bus masters to access different bus slaves simultaneously
- Providing arbitration among the bus masters when they access the same slave

Crossbar Switch (AXBS)

- Features
  - Symmetric crossbar bus switch implementation
  - Allows concurrent accesses from different masters to different slaves
  - Selectable priority for each master
  - Slave arbitration attributes configured on a slave-by-slave basis
    - Arbitration mode (Fixed or Round Robin)
    - Park Control

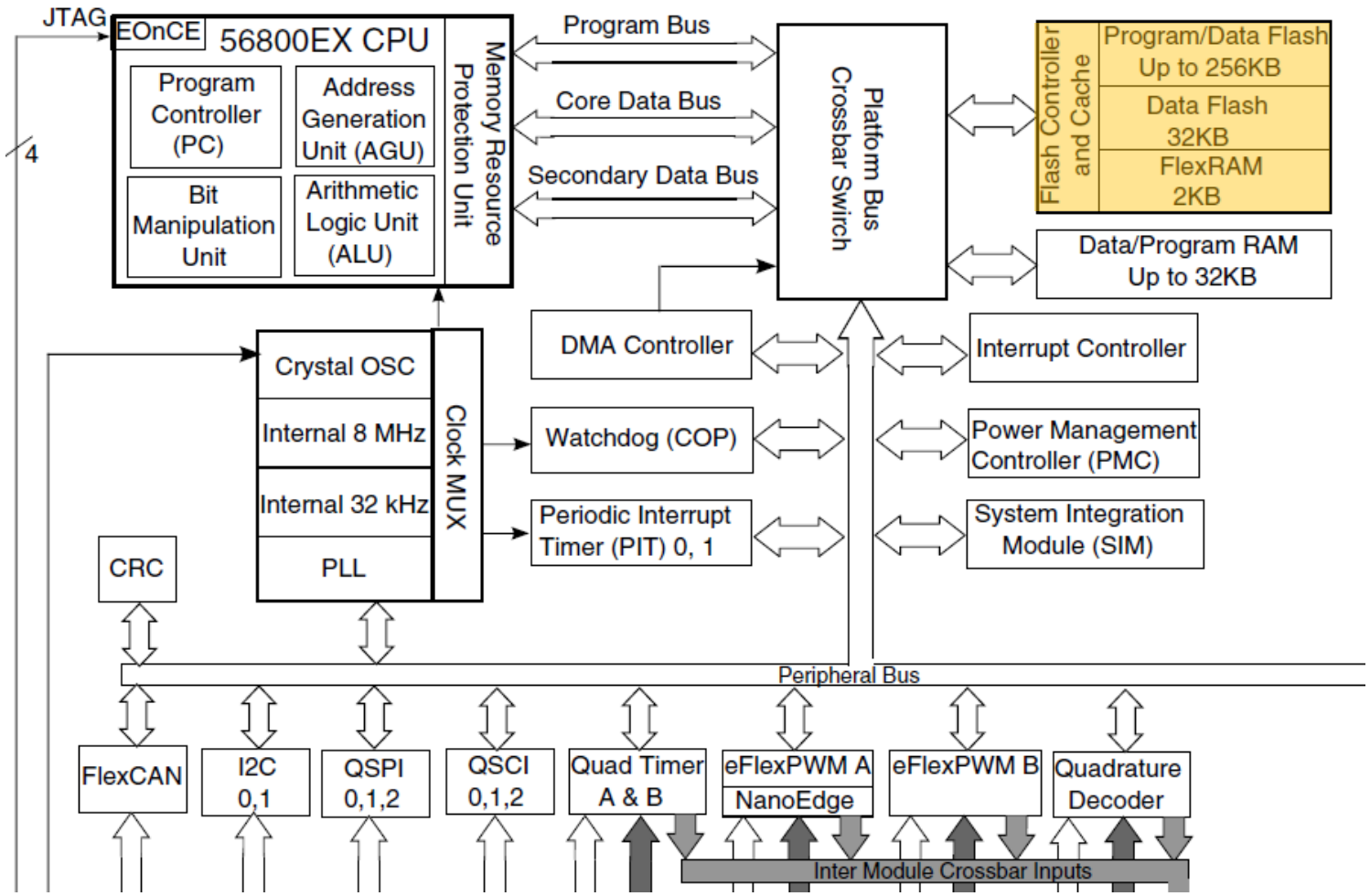
Executing code from FLASH memory

- Modern MCUs have operation clock much higher than FLASH memory clock. So the code cannot be read directly from FLASH memory
- To ensure MCU full speed operation, the reading from FLASH memory has to be accelerated by Flash Memory Controller (FMC)

56F84xx Flash Memory Controller

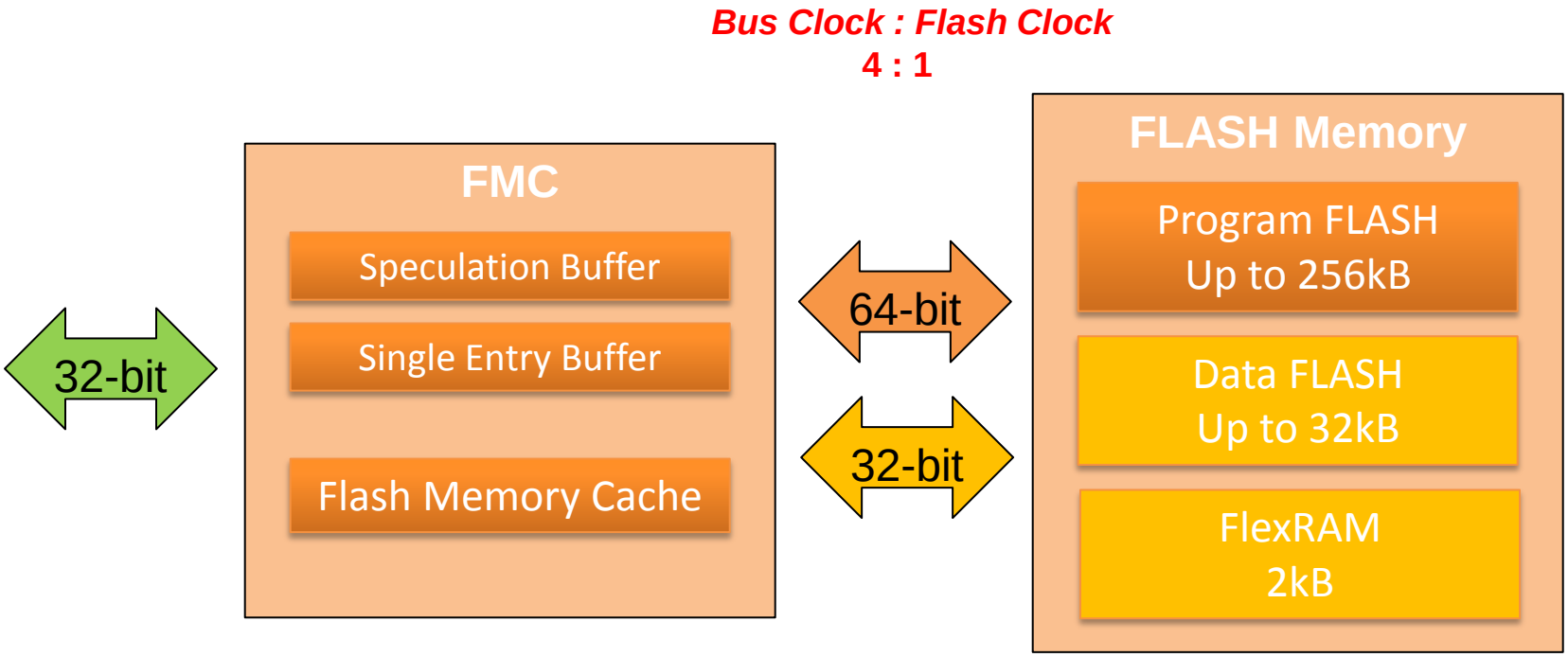
- The Flash Memory Controller (FMC) is a memory acceleration unit that provides:
  - an interface between the device and the dual-bank nonvolatile memory. Bank 0 consists of program flash memory, and bank 1 consists of FlexNVM.
  - buffers that can accelerate flash memory transfers.

56F84xx Flash Memory Controller



56F84xx Flash Memory Controller

- Principle of operation



56F84xx Flash Memory Controller

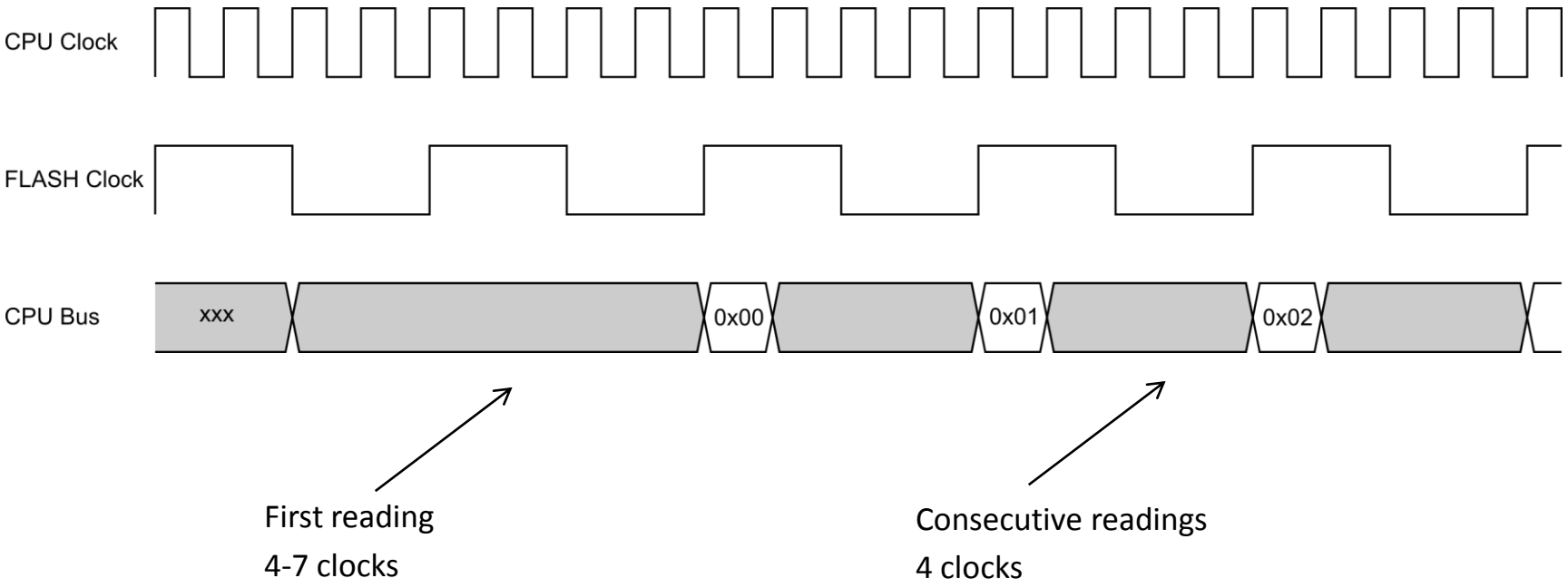
- Speculation buffer
  - The speculation buffer that reads ahead to the next word in the flash memory if there is an idle cycle.
  - Speculative pre-fetching is programmable for each bank for
  - Because many code accesses are sequential, using the speculative prefetch buffer improves performance in most cases.

56F84xx Flash Memory Controller

- Speculation buffer
  - 64-bit width
  - Supports bank 0 (program FLASH)
  - Allows full speed CPU operation for CPU/FLASH clock ratio 4:1

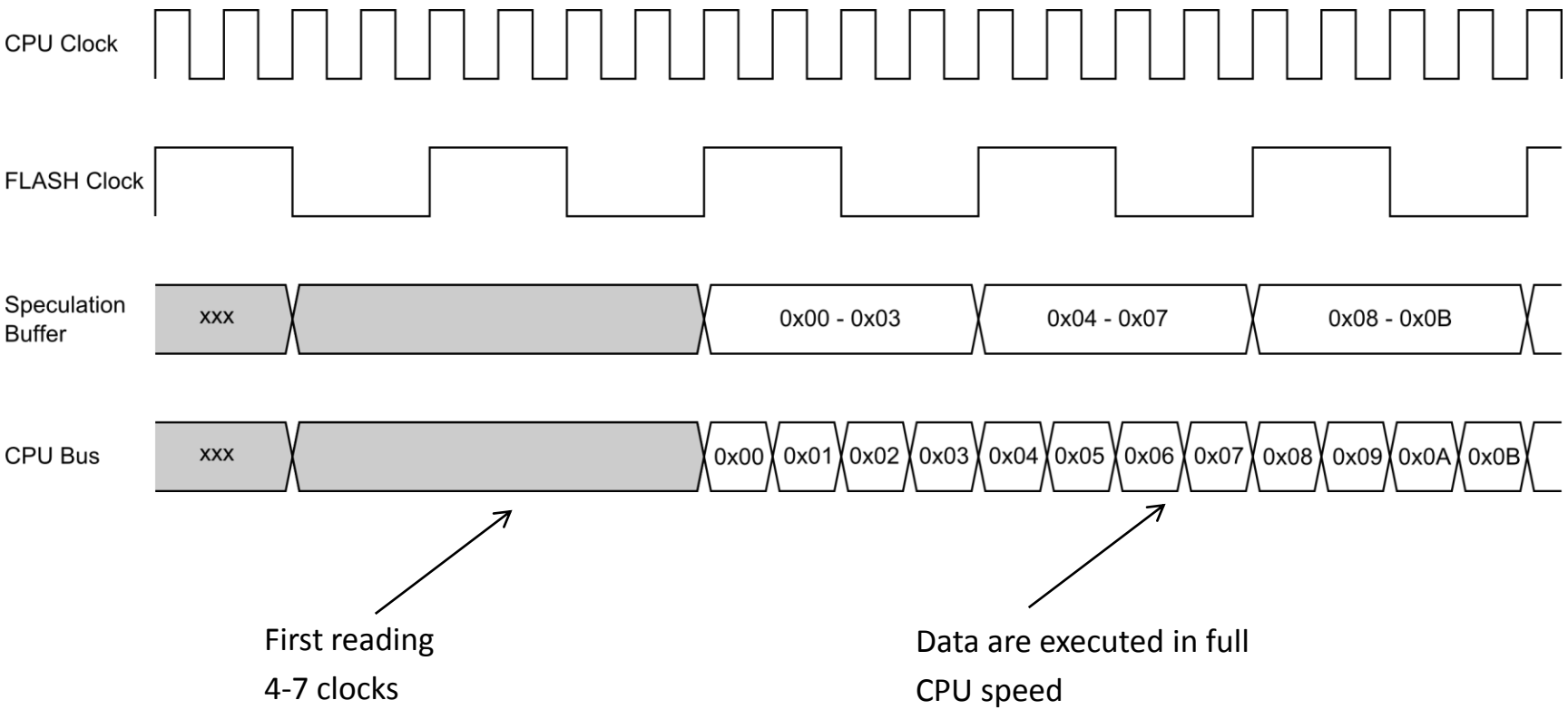
56F84xx Flash Memory Controller

- Reading from Flash without Speculation Buffer



56F84xx Flash Memory Controller

- Reading from Flash with Speculation Buffer



56F84xx Flash Memory Controller

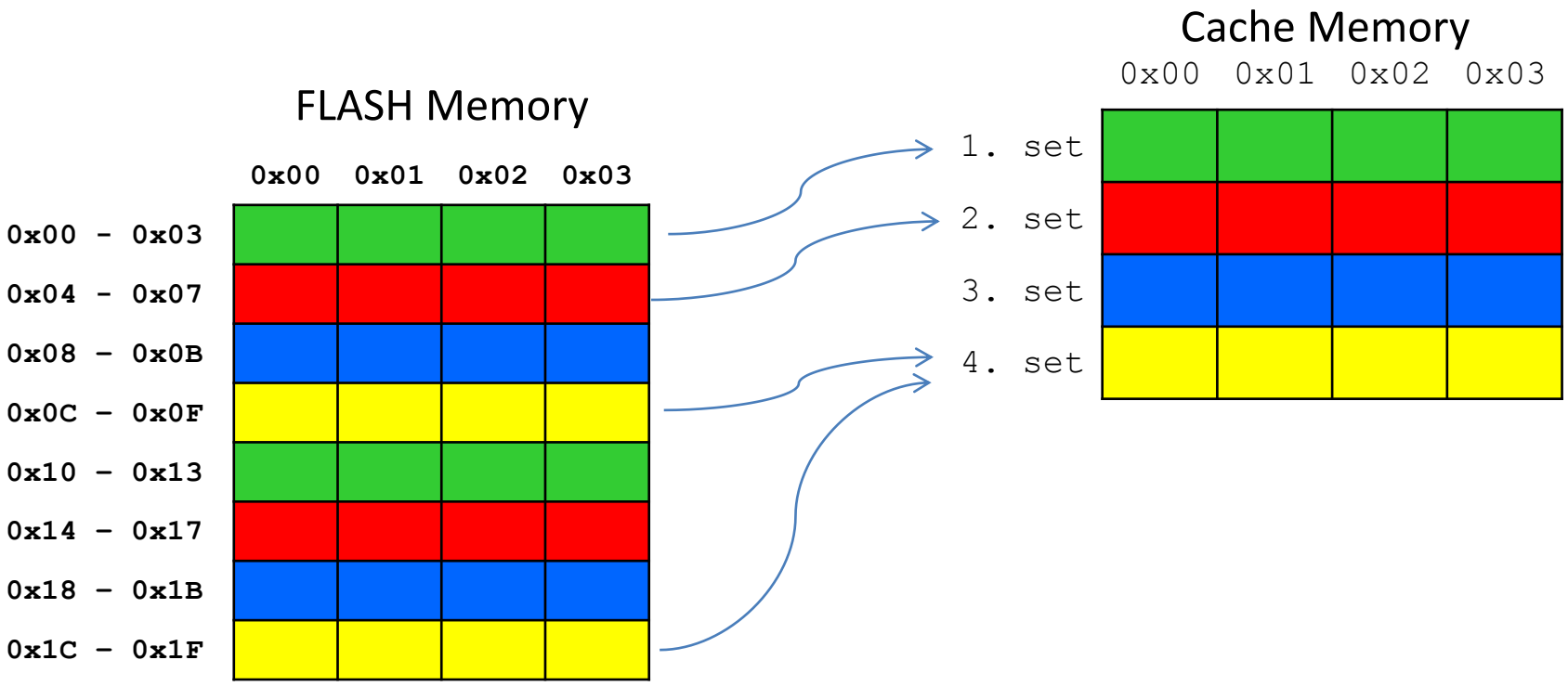
- Flash Memory Cache
  - 64-bit, 4-way associative cache, 8 sets
  - Controls for replacement algorithm
  - Lock per way
  - Cache allows fast access to previously executed code

56F84xx Flash Memory Controller

- Direct Cache Memory
  - Flash memory is divided into sets according to cache set size
  - Each Flash memory set has dedicated place in cache memory
  - The new flash memory data always re-write corresponding cache set

56F84xx Flash Memory Controller

- Direct Cache Memory

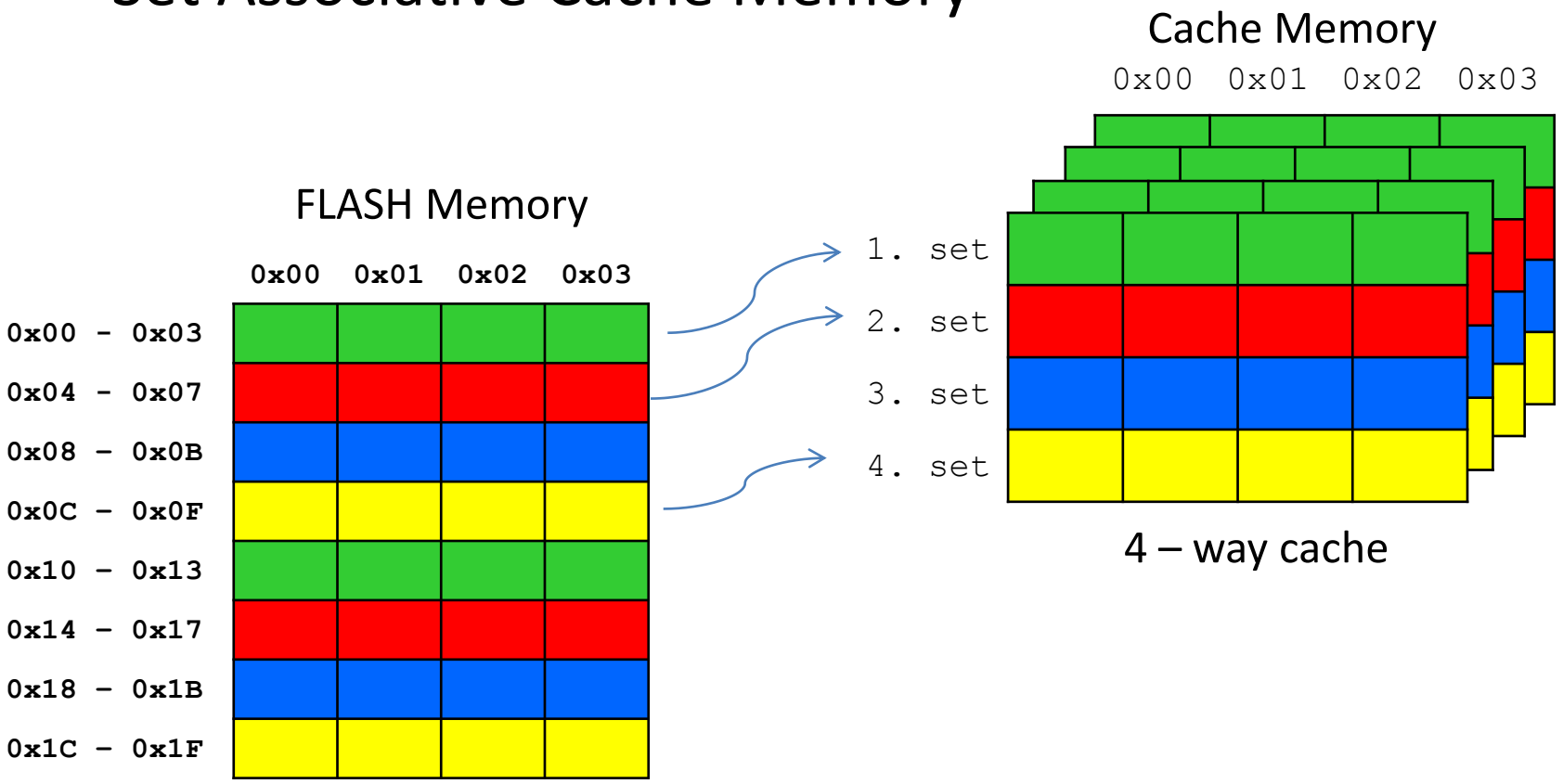


56F84xx Flash Memory Controller

- Set Associative Cache Memory
  - Flash memory is divided into sets according to cache set size
  - Each Flash memory set has dedicated place in cache memory in every cache way
  - The new flash memory data always re-write corresponding cache set

56F84xx Flash Memory Controller

- Set Associative Cache Memory

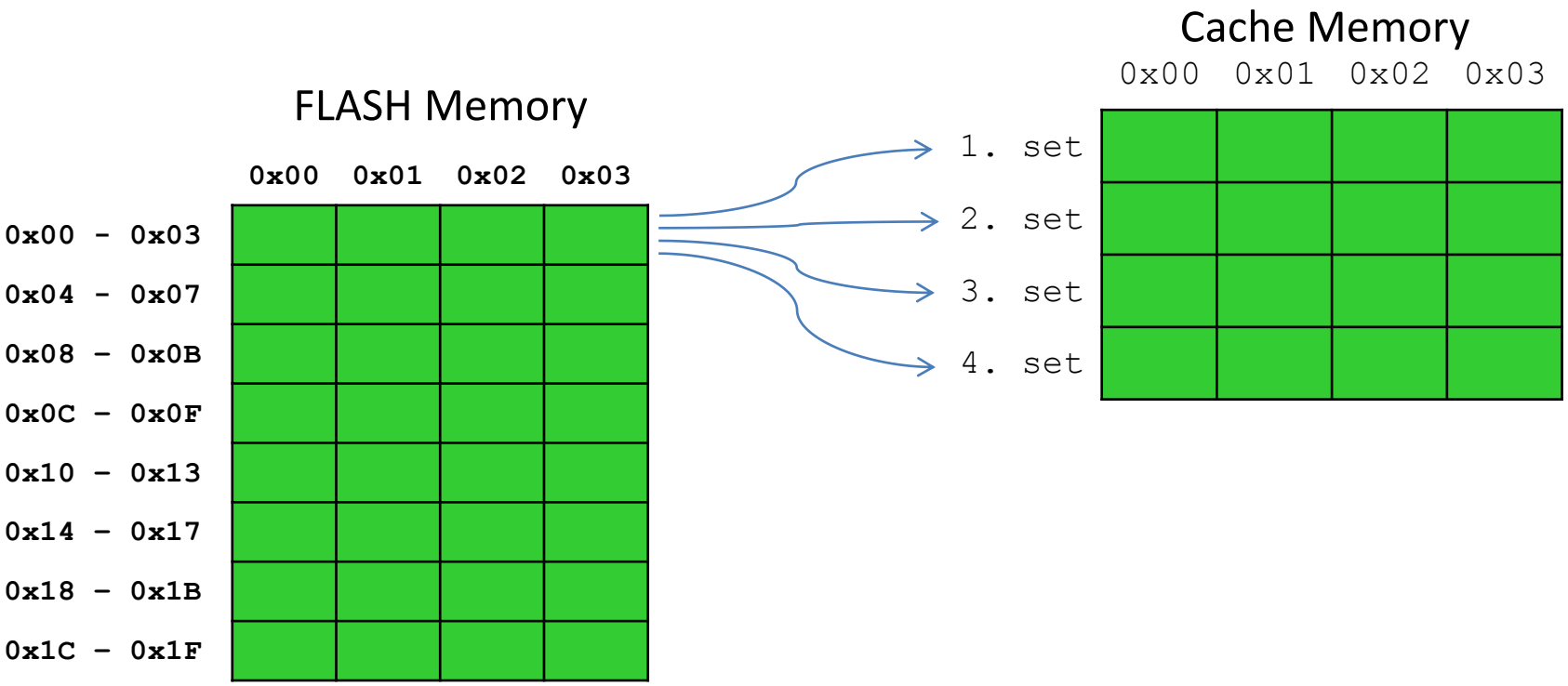


56F84xx Flash Memory Controller

- Fully Associative Cache Memory
  - Flash memory is divided into sets according to cache set size
  - Each Flash memory set can be placed in cache set

56F84xx Flash Memory Controller

- Fully Associative Cache Memory



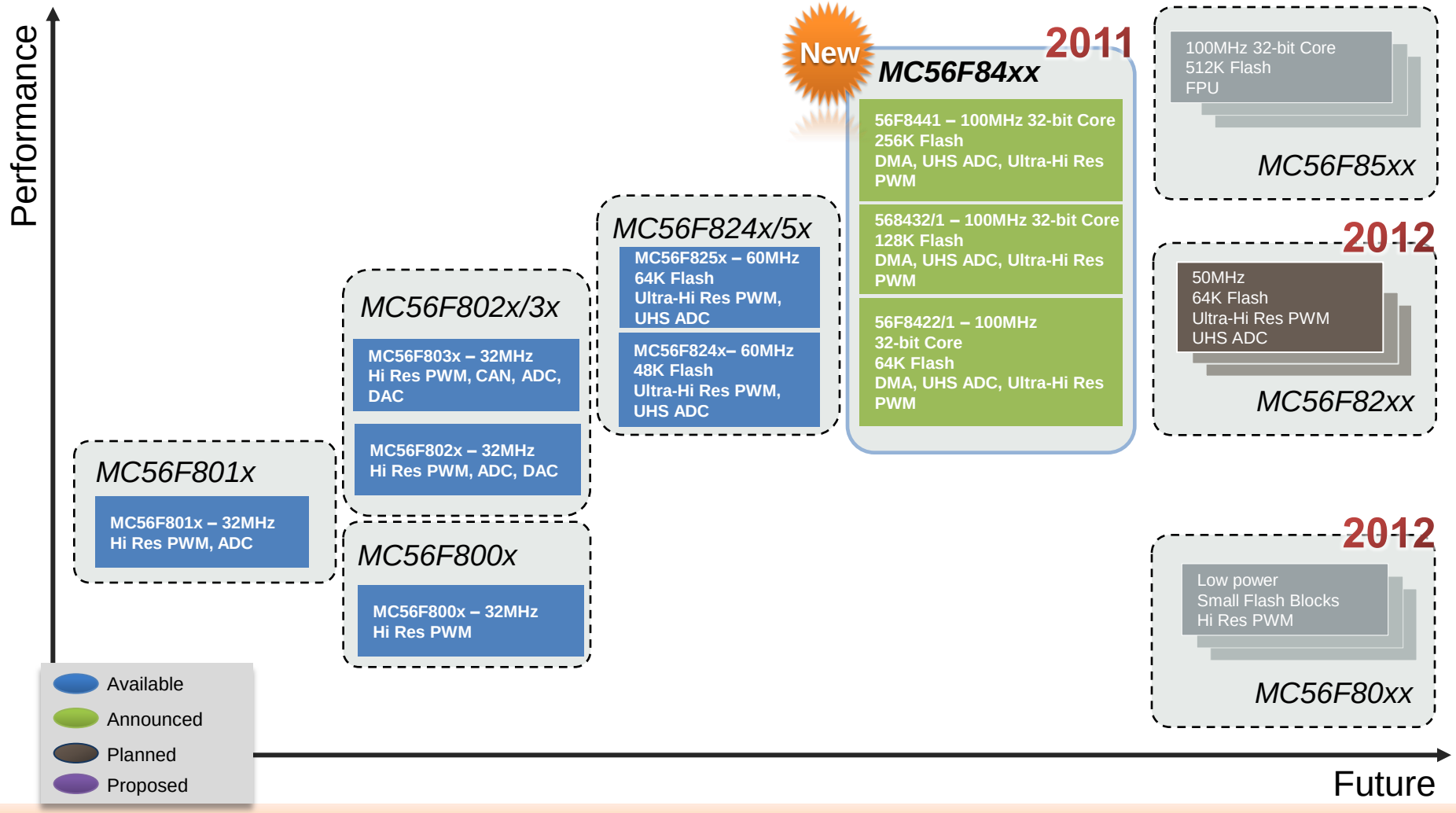
56F84xx Flash Memory Controller

- Single Entry Buffer
  - Single Entry Buffer holds previously accessed data (it is in fact single set cache)
  - It is used when CPU bus is shorter than Flash memory interface
  - Either cache or single entry buffer can be enabled

Crossbar and FMC Conclusion

- It is very important to understand functionality of Crossbar switch and Flash memory interface since their settings have impact on the performance of the MCU
- Improper setting may impact deterministic behavior in time critical applicaitons

DSC Roadmap


Performance
New
2011
MC56F84xx
56F8441 – 100MHz 32-bit Core
256K Flash
DMA, UHS ADC, Ultra-Hi Res PWM
568432/1 – 100MHz 32-bit Core
128K Flash
DMA, UHS ADC, Ultra-Hi Res PWM
56F8422/1 – 100MHz
32-bit Core
64K Flash
DMA, UHS ADC, Ultra-Hi Res PWM
100MHz 32-bit Core
512K Flash
FPU
MC56F85xx
2012
50MHz
64K Flash
Ultra-Hi Res PWM
UHS ADC
MC56F82xx
MC56F824x/5x
MC56F825x – 60MHz
64K Flash
Ultra-Hi Res PWM, UHS ADC
MC56F824x– 60MHz
48K Flash
Ultra-Hi Res PWM, UHS ADC
MC56F802x/3x
MC56F803x – 32MHz
Hi Res PWM, CAN, ADC, DAC
MC56F802x – 32MHz
Hi Res PWM, ADC, DAC
MC56F801x
MC56F801x – 32MHz
Hi Res PWM, ADC
MC56F800x
MC56F800x – 32MHz
Hi Res PWM
2012
Low power
Small Flash Blocks
Hi Res PWM
MC56F80xx
Available
Announced
Planned
Proposed
Future

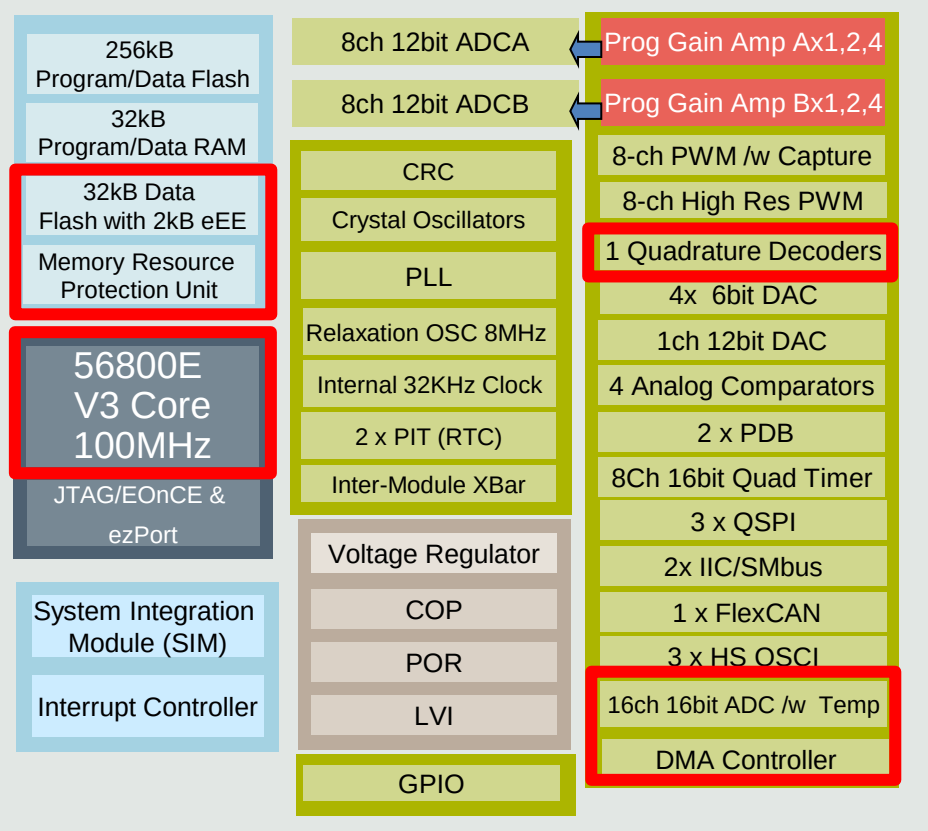
MC56F84xx Features

100 MHz/100MIPS 56800 V3 Core
- Harvard architecture
- **32 x 32bit MAC and 32bit arithmetic operation**

- 2.7-3.6V Operation

- 256kB Program/Data FLASH
- 32kB Data Flash with up to 2kB of eEE
- 32kB Data/Program RAM
- Resource Protection Unit

- 3 HS-QSCI (8MBS) , 3xQSPI, 2xIIC/SMBus, 1xFlexCAN
- Multi-purpose timers
  - 2 Periodic Timers with Real Time Interrupt Generation
  - 2 Programmable Delay Blocks
  - 8Ch multifunction timers
- 8ch High Resolution PWM Channels
  - **312ps PWM and PFM resolution**
- 8ch PWM Channels with Input Capture
- 8ch x 2 12-bit ADC converter with built-in PGA
  - **300ns/3.33Msps conversion time with 12bit resolution**
- 8ch 16bit SAR ADC with built-in temperature sensor and band gap.
  - 2us conversion time.
- 4 Analog Comparators
- 1 Quadrature Decoder
- 1ch 12bit DAC with external outputs + 4ch 6bit DAC
- DMA controller
- Inter-Module Crossbar
- On-chip voltage regulator (Single 3.3V Power Supply)
- System Integration : Internal relaxation oscillator, PLL, COP, 32kHz , EWM, auxiliary Internal clock, low voltage detect, EZPort
- 5V tolerant I/O
- Temperature Range: -40°C to +105°C



48 LQFP, 64 LQFP, 80LQFP, 100LQFP

56F8400 Series Feature Summary

	Fully Featured		Digital Control		Digital Control						Dual Motor				Single Motor			
Part Number																		
Core MHz	100	100	100	100	100	100	100	100	100	100	80	80	80	80	60	60	60	60
Flash Mem (kB)	256	256	128	128	128	128	64	64	128	128	256	256	128	128	128	128	64	64
SRAM Mem (kB)	32	32	24	24	24	24	16	16	24	24	32	32	16	16	16	16	8	8
Data Flash / EE Mem (kB)	32/2	32/2	32/2	32/2	32/2	32/2	32/2	32/2	0	0	32/2	32/2	32/2	32/2	0	0	0	0
Cyc ADC Chn	2x8	2x8	2x8	2x8	2x8	2x8	2x8	2x8	2x5	2x5	2x8	2x8	2x8	2x8	2x8/2x5	2x8/2x5	2x8/2x5	2x8/2x5
SAR ADC Chn	1x16	1x16	1x16	1x16	0	0	0	0	0	0	0	0	0	0	0	0	0	0
PWM uE Chn	8	8	8	8	8	8	8	8	8	8	8	8	8	8	0	0	0	0
PWM stnd Chn	8	8	8	8	0	0	0	0	0	0	8	8	8	8	8	8	8	8
DAC	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
Quad Decoder	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1
DMA	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CMP	4	4	3	3	3	3	2	2	2	2	3	3	3	3	2	2	2	2
QSCI	3	3	3	3	3	3	2	2	2	2	3	3	2	2	2	2	2	2
QSPI	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2
I2C	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
FlexCAN	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0
Package	100	80	100	80	80	64	80	64	64	48	100	80	100	80	64	48	64	48

DSP56800E Version 3 Core Improvement

(the differences between V2 core and V3 core)

New Instructions

- 32 x 32 -> 32/64 Multiply and MAC Instructions
  - ✓ IMAC32 - Integer Multiply-Accumulate 32 bits x 32 bits -> 32 bits
  - ✓ IMPY32 - Integer Multiply 32 bits x 32 bits -> 32 bits
  - ✓ IMPY64 - Integer Multiply 32 bits x 32 bits -> 64 bits
  - ✓ IMPY64UU - Unsigned Integer Multiply 32 bits x 32 bits -> 64 bits
  - ✓ MAC32 - Fractional Multiply-Accumulate 32 bits x 32 bits -> 32 bits
  - ✓ MPY32 - Fractional Multiply 32 bits x 32 bits -> 32 bits
  - ✓ MPY64 - Fractional Multiply 32 bits x 32 bits -> 64 bits
- Multi-Bit Clear-Set instruction to improve flexibility of peripheral register handling.

Other Features

- Bit Reversed Address Mode For FFT algorithms.
- Swap all address generation Unit Registers with Shadowed registers to reduce Interrupt context switch latency.

New System Peripherals

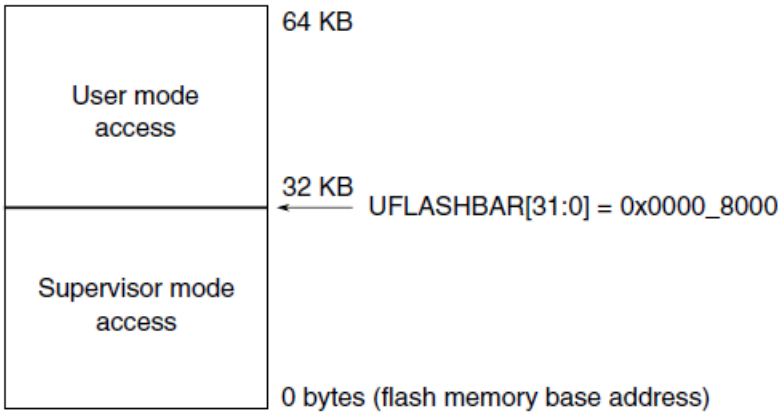
- Crossbar Switch (AXBS)
- FLASH Memory Controller (FMC)
- Memory Resource Protection (MRP)
- DMA Controller
- AND/OR/INVERT (AOI) Module
- Inter-Peripheral Crossbar Switch A (XBARA)
- Inter-Peripheral Crossbar Switch B (XBARB)

Memory Resource Protection (MRP)

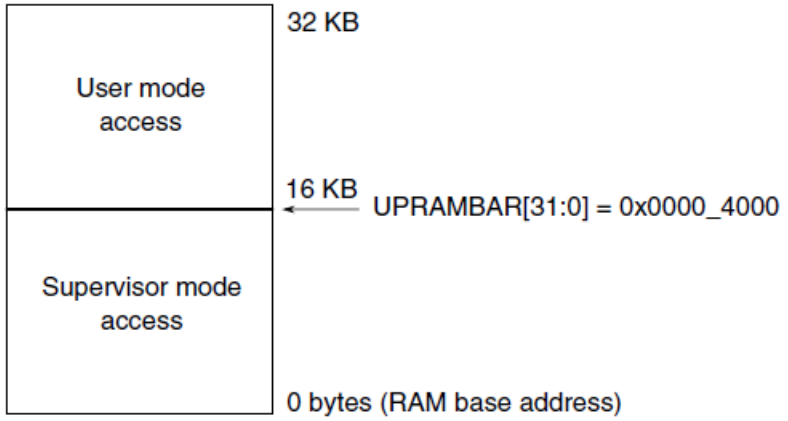
- Partition software into two modes: supervisor software and user software
- Partition system address spaces and resources, both code and data, into two modes
  - Flash memory regions defined with 8 KB granularity
  - RAM memory regions defined with 512 byte granularity
- Provide secure methods of transfer between modes
- Provide separate stacks and stack pointers for supervisor and user modes

Memory Resource Protection (MRP)

- Example of MPR setting



Total flash size: 64kB



Total RAM size: 64kB

DMA Controller

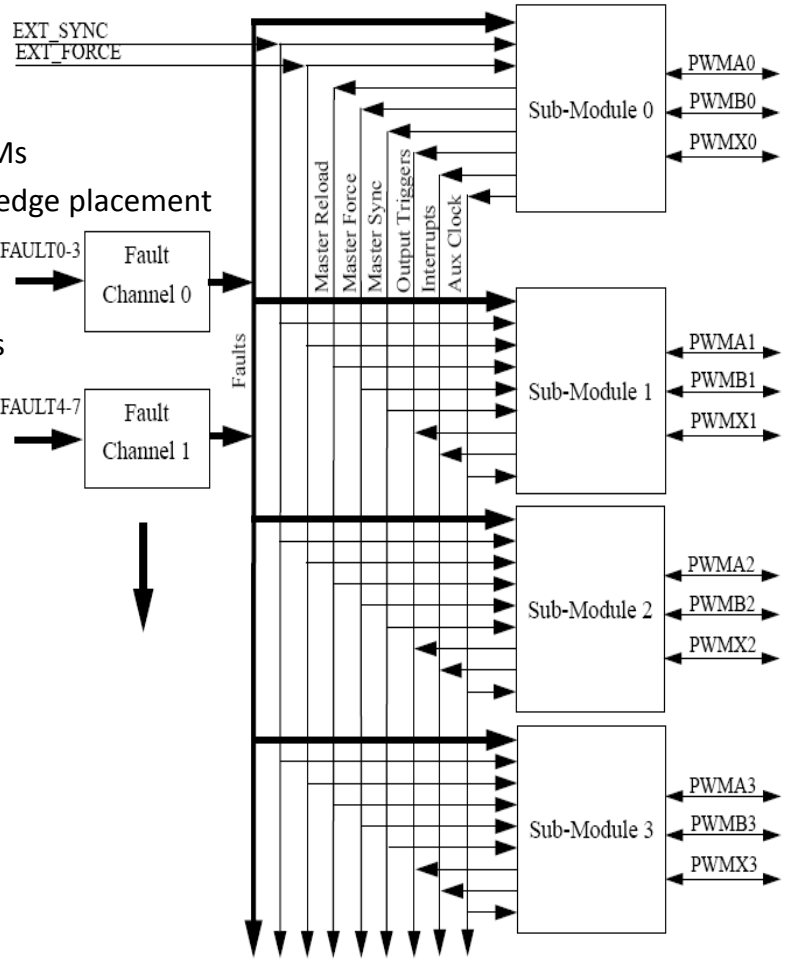
- Four independently programmable DMA controller channels
- Dual-address transfers via 32-bit master connection to the system bus
- Data transfers in 8-, 16-, or 32-bit blocks
- Continuous-mode or cycle-steal transfers from software or peripheral initiation
- One programmable input selected from 16 possible peripheral requests per channel
- Automatic hardware acknowledge/done indicator from each channel
- Independent source and destination address registers
- Optional modulo addressing and automatic updates of source and destination addresses
- Independent transfer sizes for source and destination
- Optional auto-alignment feature for source or destination accesses
- Optional automatic single or double channel linking
- Programming model accessed via 32-bit slave peripheral bus
- Channel arbitration on transfer boundaries using fixed priority scheme

New Peripherals

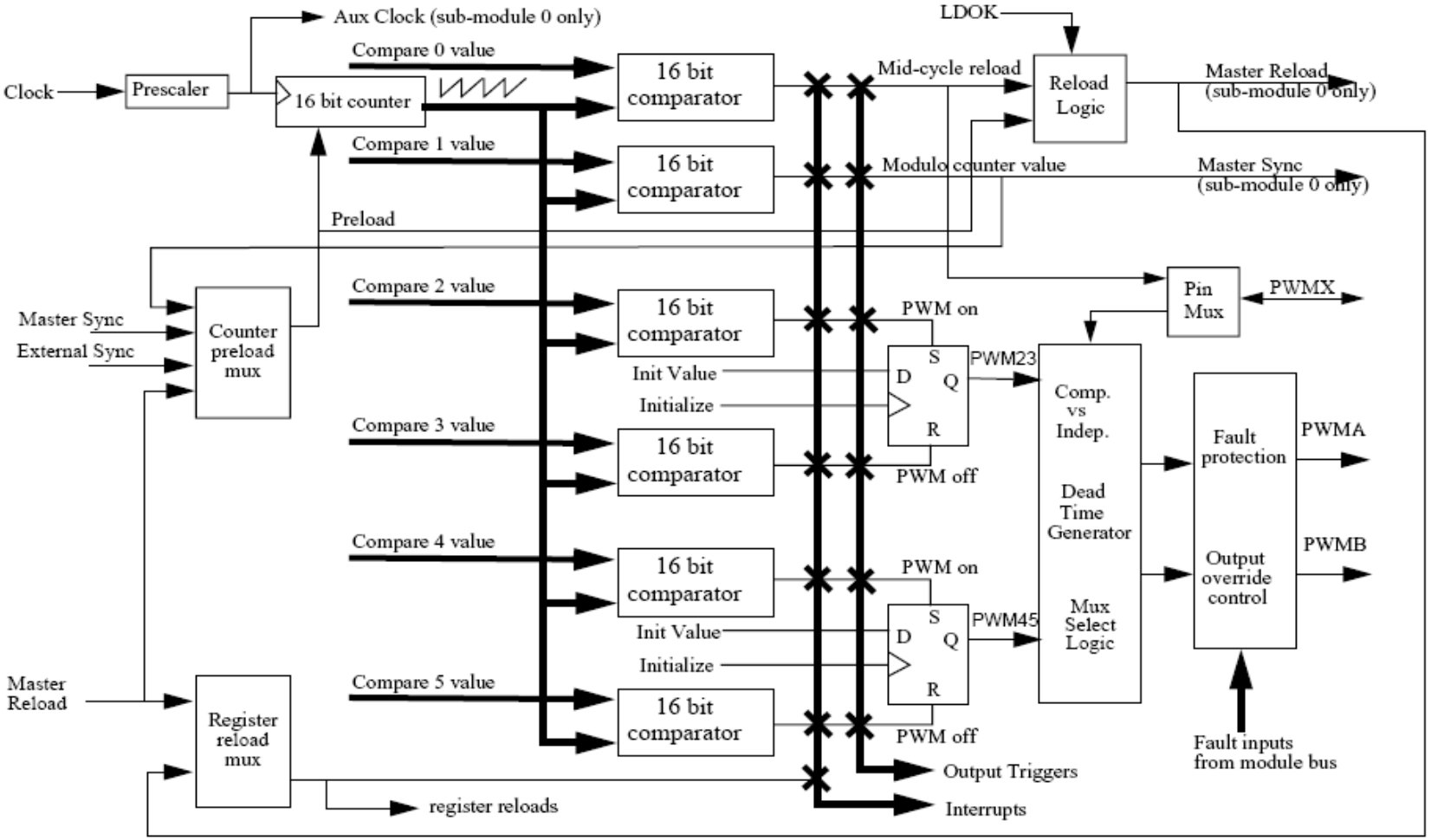
- Cyclic Redundancy Check (CRC)
- 16-bit SAR Analog-to-Digital Converter (ADC16)
- Enhanced Flexible Pulse Width Modulator (PWMA, PWMB)
- Quadrature Decoder (ENC)

Enhanced Flex Pulse Width Modulator (eFlexPWM)

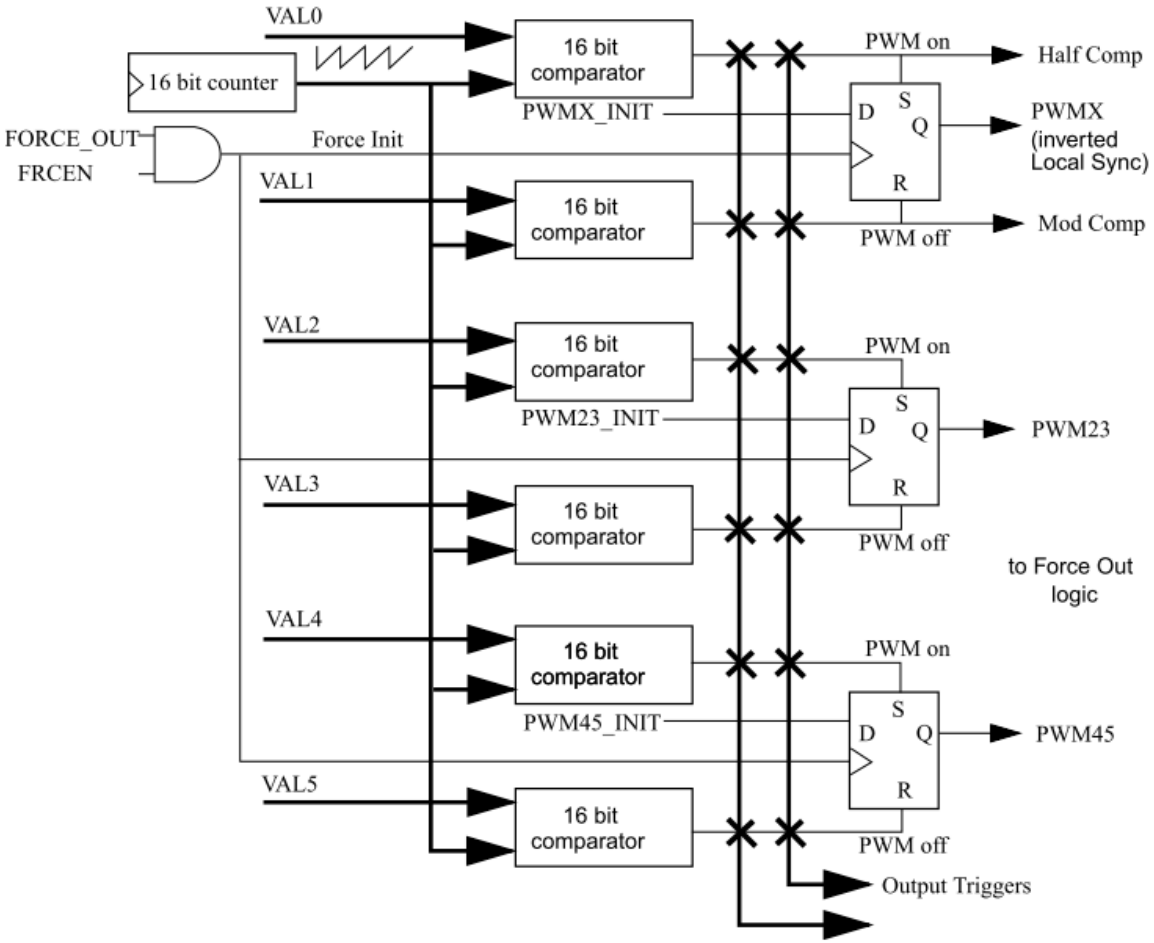
- Four independent sub-modules with own time base, two PWM outputs + 1 auxiliary PWM input/output
- 16 bits resolution for center, edge aligned, and asymmetrical PWMs
- Fractional delay for enhanced resolution of the PWM period and edge placement
- Complementary pairs or independent operation
- Independent control of both edges of each PWM output
- Synchronization to external hardware or other PWM sub-modules
- Double buffered PWM registers
- Integral reload rates from 1 to 16 include half cycle reload
- Half cycle reload capability
- Multiple output trigger events per PWM cycle
- Support for double switching PWM outputs
- Fault inputs can be assigned to control multiple PWM outputs
- Programmable filters for fault inputs
- Independently programmable PWM output polarity
- Independent top and bottom deadtime insertion
- Individual software control for each PWM output
- Software control, and swap features via FORCE_OUT event
- Compare/capture functions for unused PWM channels
- Enhanced dual edge capture functionality



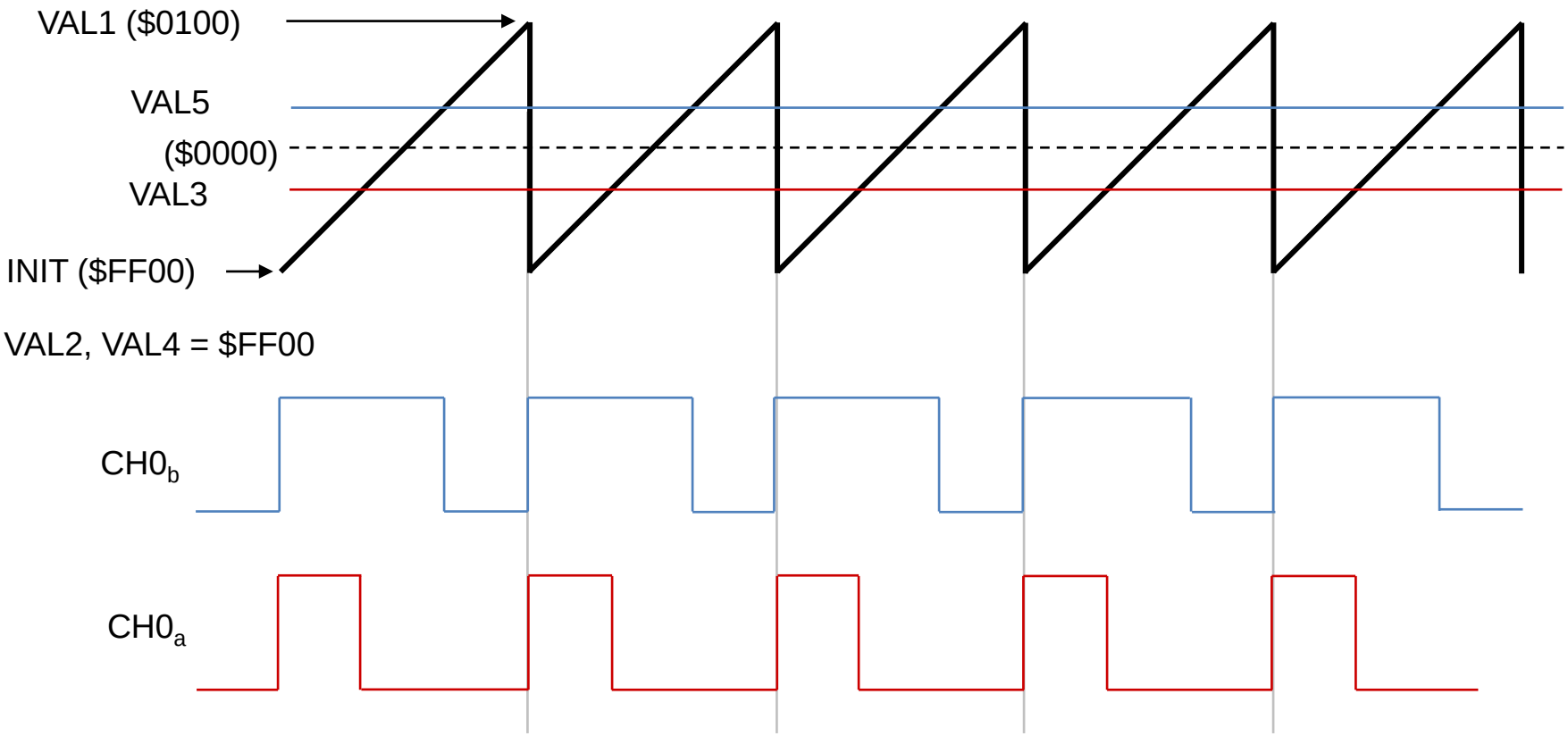
eFlexPWM - Sub-Module Detail



eFlexPWM – PWM Generation

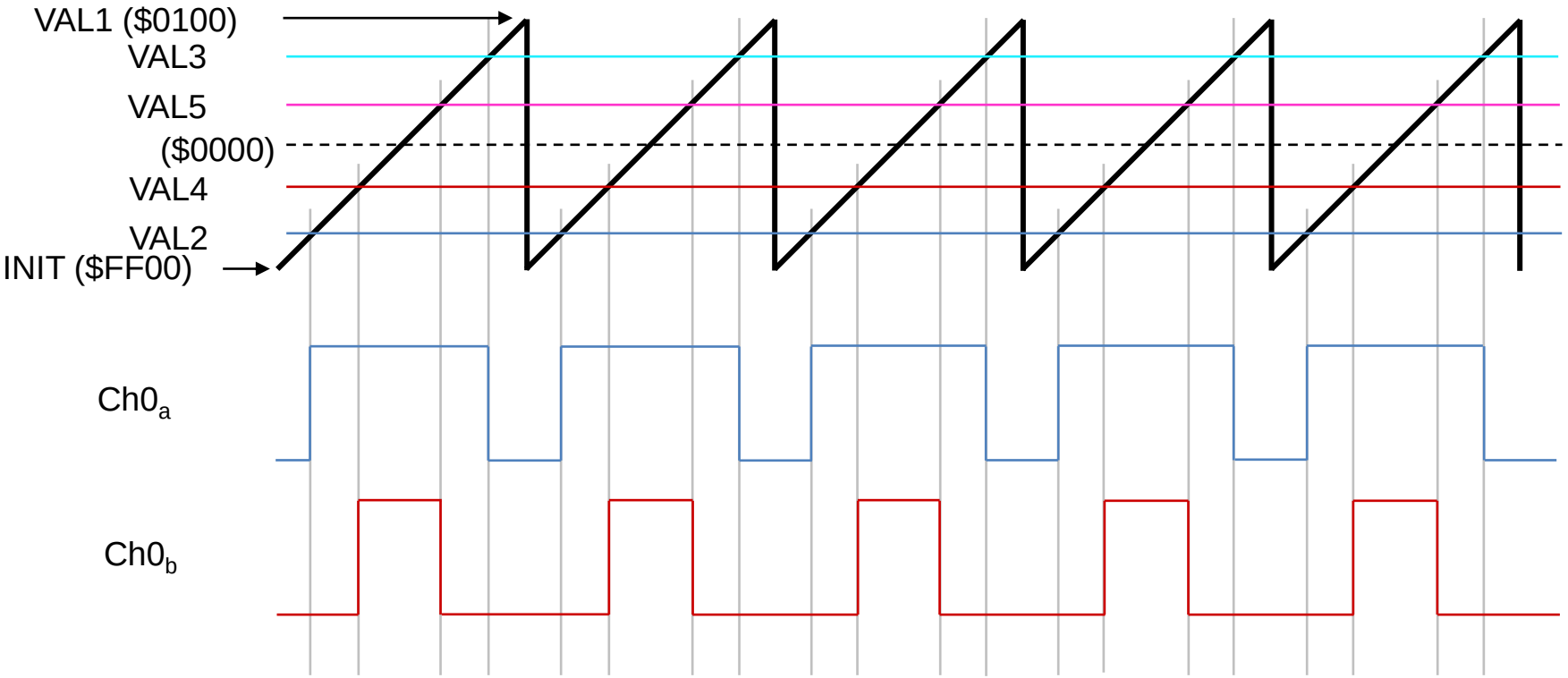


eFlexPWM – Edge Aligned PWM Generation



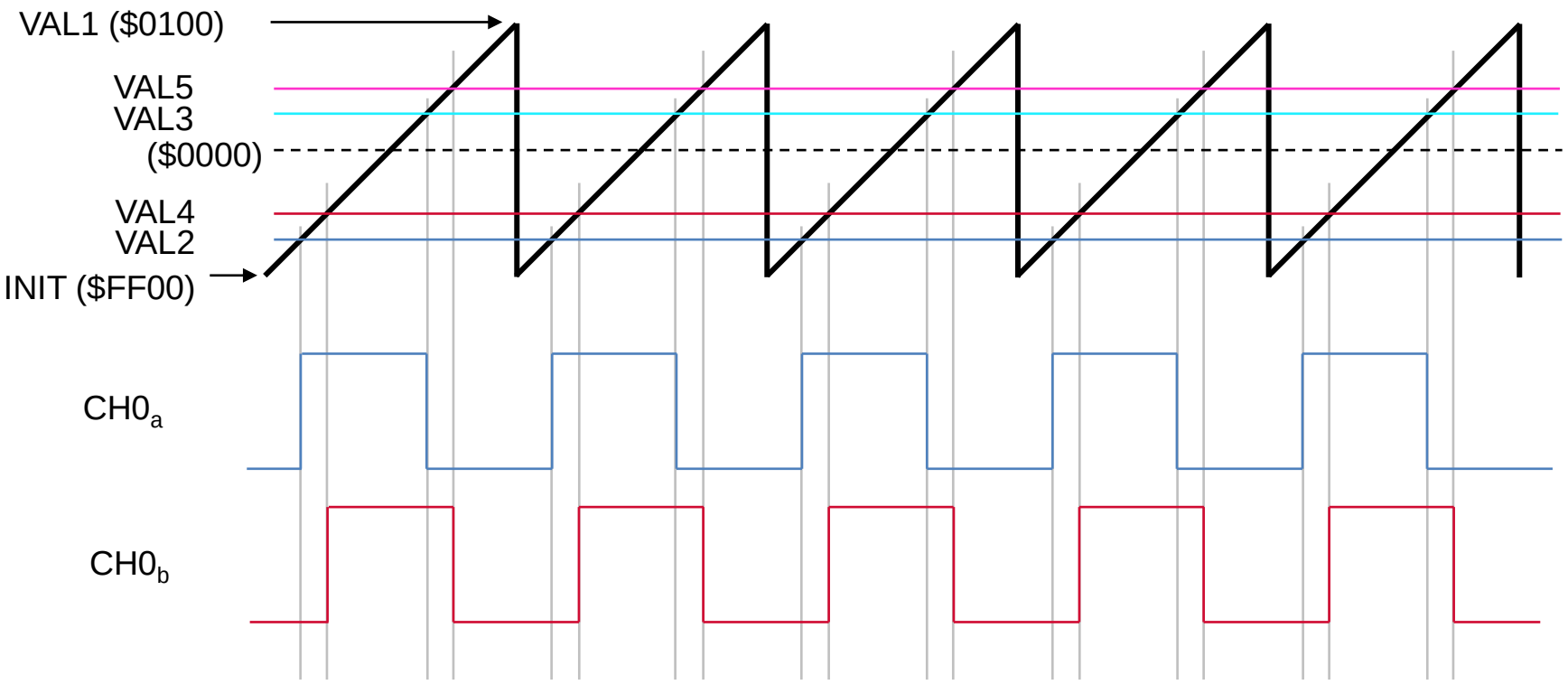
- All PWM-on values are set to the init value, and never changed again. Positive PWM-off values generate pulse widths above 50% duty cycle. Negative PWM-off values generate pulse widths below 50% duty cycle . This works well for bipolar waveform generation.

eFlexPWM – Center Aligned PWM Generation



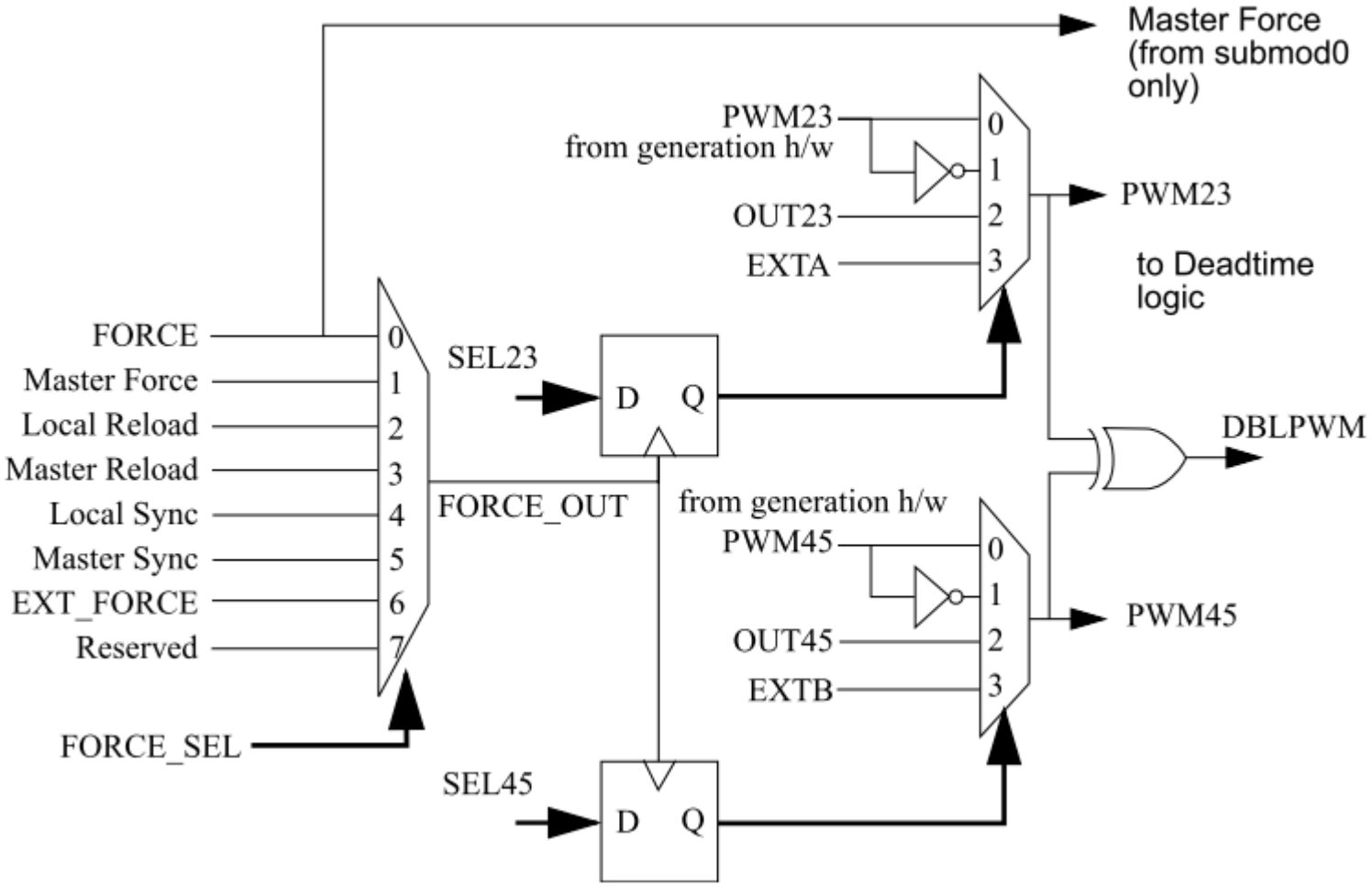
- When the Init value is the signed negative of the Modulus value, the PWM module works in signed mode. Center-aligned operation is achieved when the turn-on and turn-off values are the same number, but just different signs.

eFlexPWM – Shifted PWM Generation

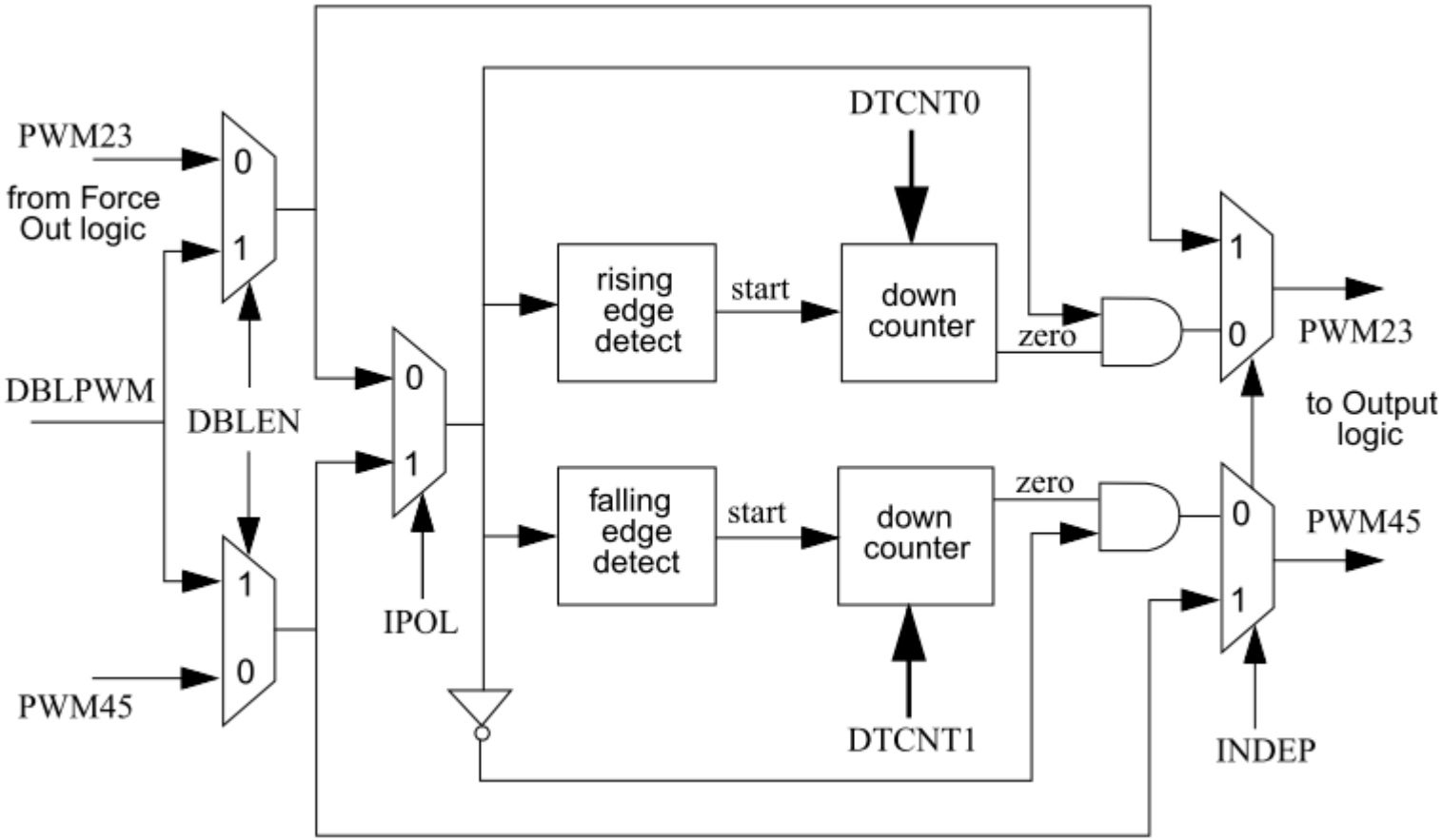


- In this example, both PWMs have the same duty-cycle. However, the edges are shifted relative to each other by simply biasing the compare values of one waveform relative to the other.

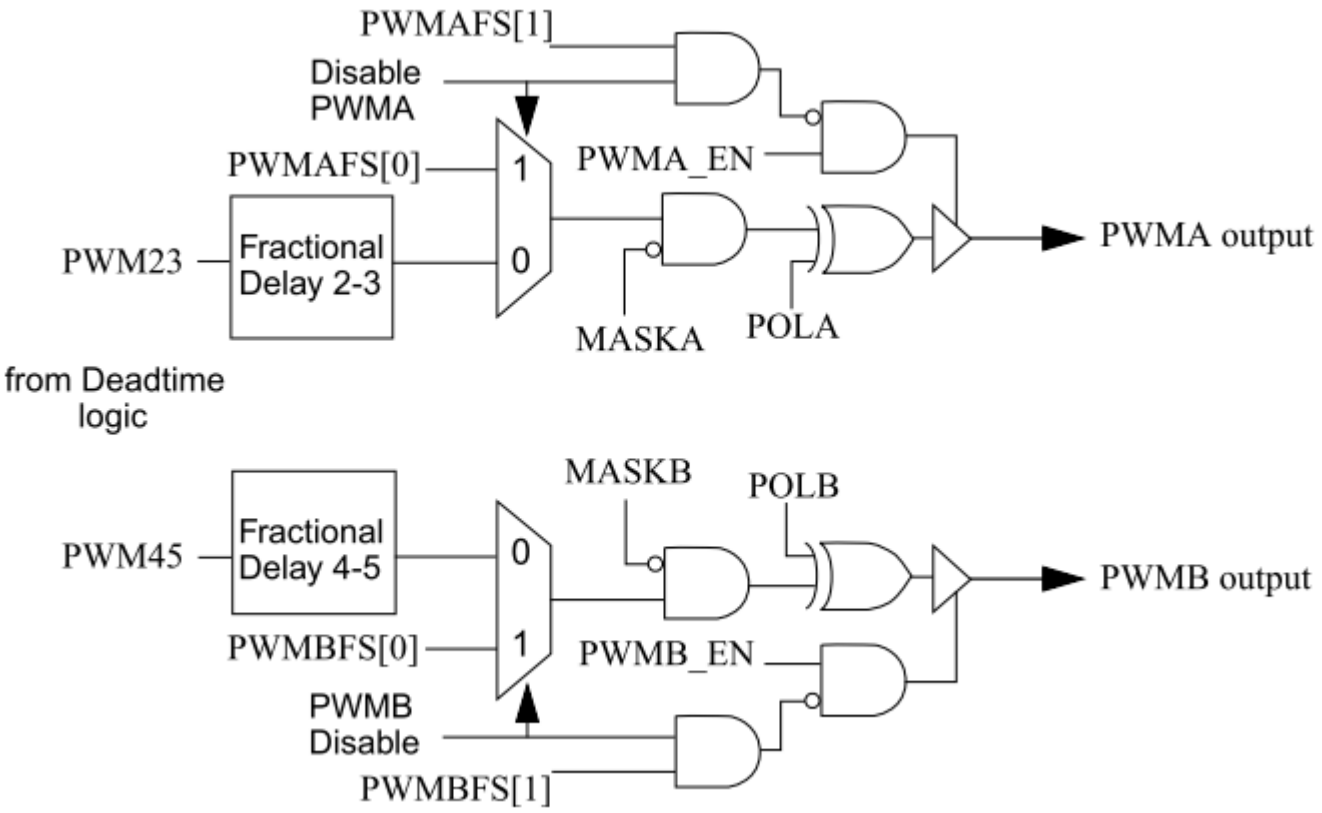
eFlexPWM – Force Output Logic



eFlexPWM – Complementary and Deadtime Logic

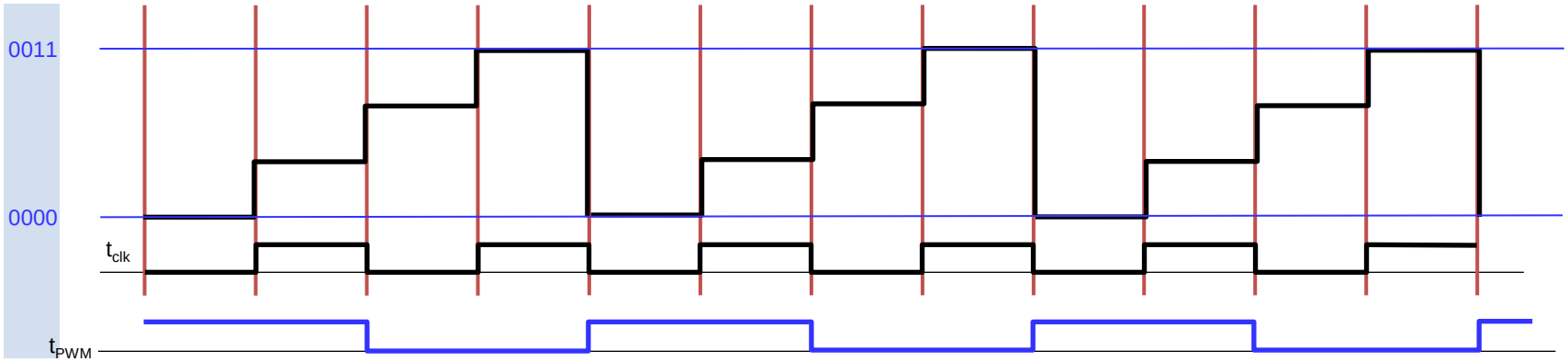


eFlexPWM – Fractional Delay and Output Logic



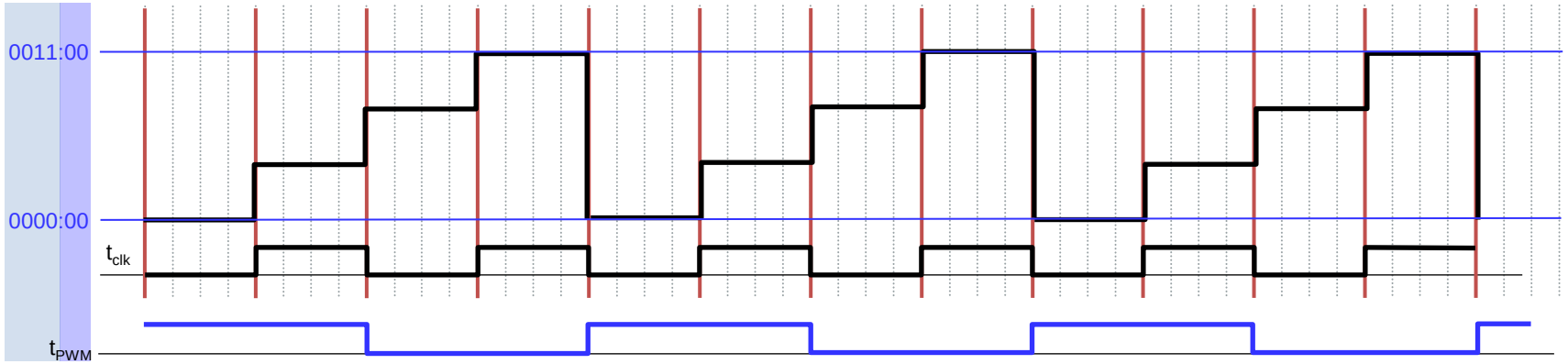
eFlexPWM – High Resolution Duty Cycle Generation

- PWM resolution is given by input clock of PWM module



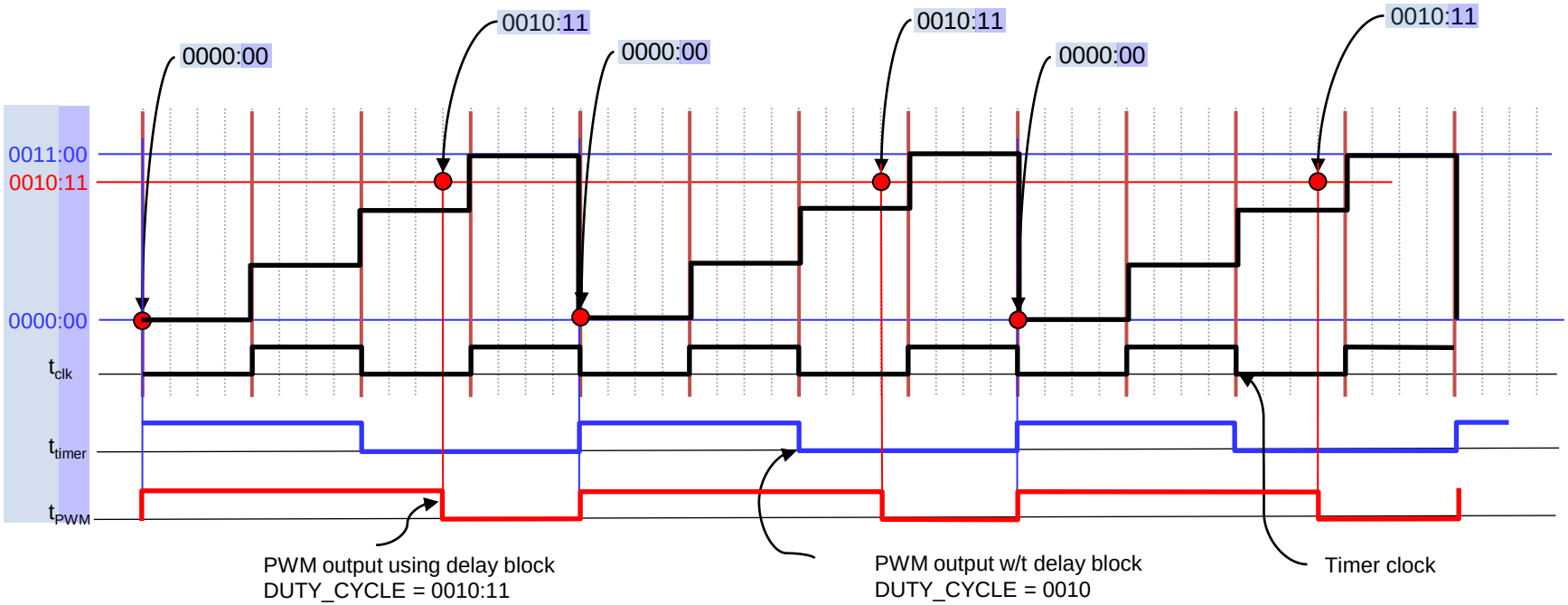
eFlexPWM – High Resolution Duty Cycle Generation

- PWM resolution is given by input clock of PWM module
- The PWM resolution can be enhanced by analog delay circuit, which can place edge between two edges, derived from input clock
- *Example*
  - Consider 2-bit analog delay block
  - Let's generate PWM signal with MODULO=4:0, DUTY_CYCLE=2:3 (68.75 %)



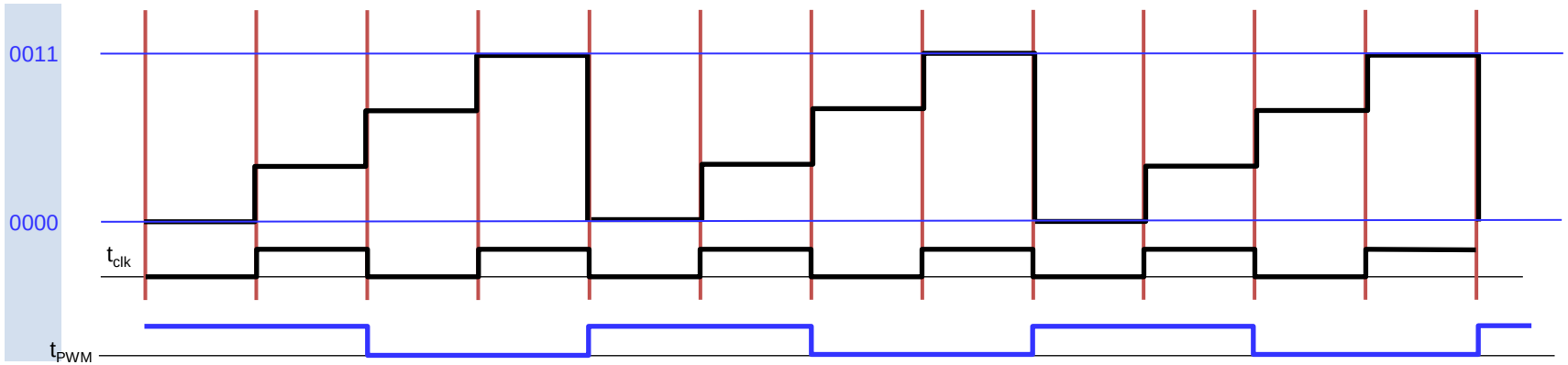
eFlexPWM – High Resolution Duty Cycle Generation

- At high resolution duty cycle generation
  - The leading edge is usually aligned with digital clock
  - The falling edge is generated by delay block
  - The analog delay is constant every PWM period



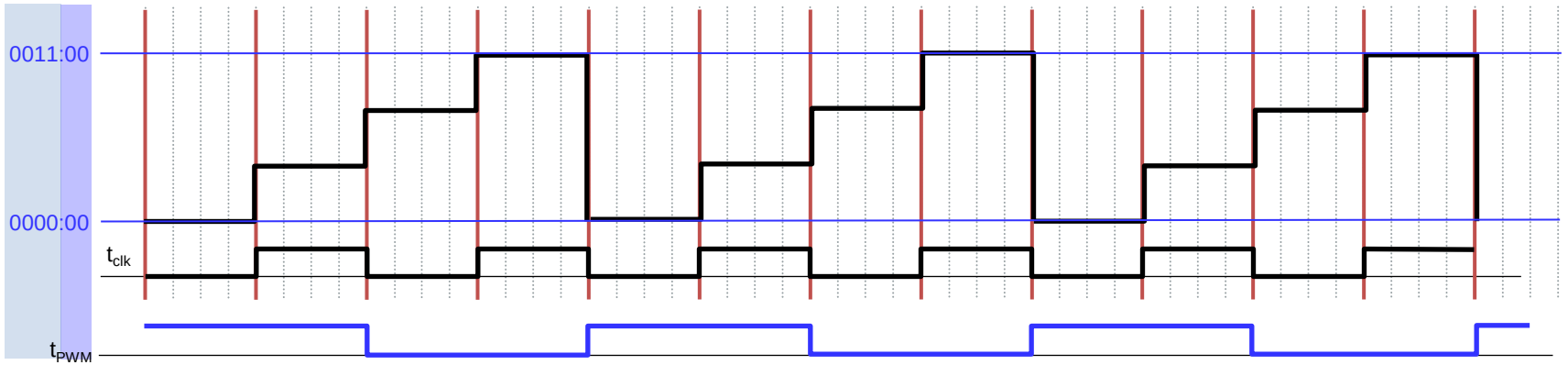
eFlexPWM – High Resolution Frequency Generation

- *Example*
  - Consider 2-bit analog delay block
  - Let's generate PWM signal with MODULO=4:2, DUTY_CYCLE=2:1 (50 %)



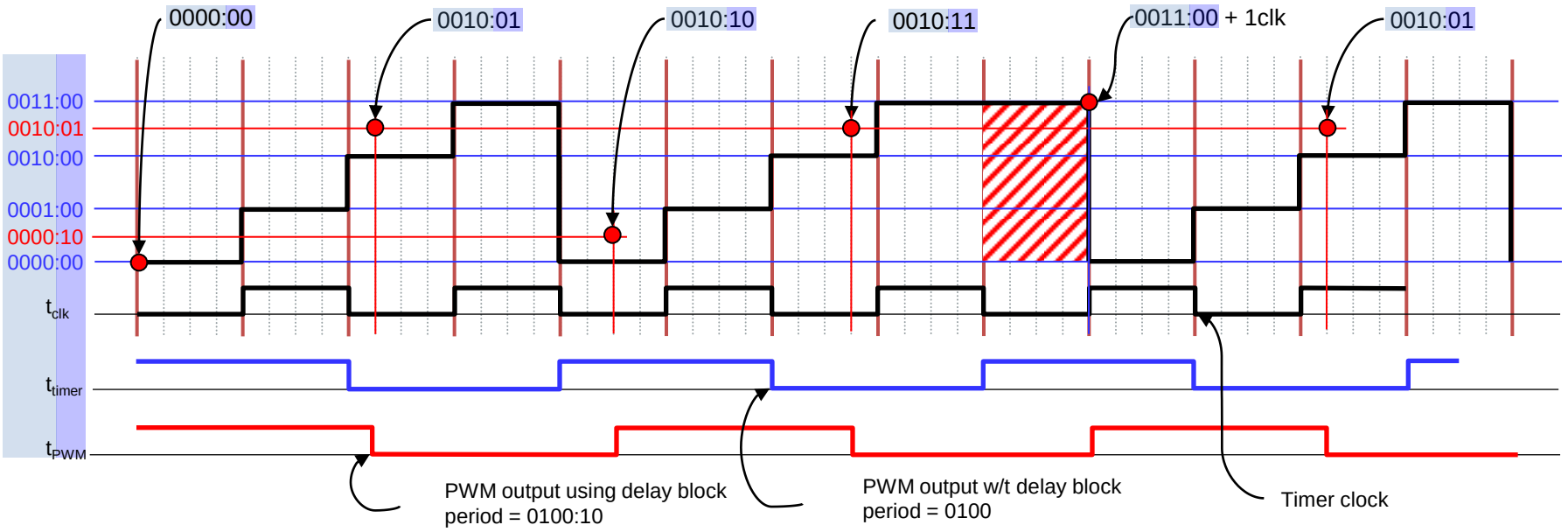
eFlexPWM – High Resolution Frequency Generation

- *Example*
  - Consider 2-bit analog delay block
  - Let's generate PWM signal with MODULO=4:2, DUTY_CYCLE=2:1 (50 %)



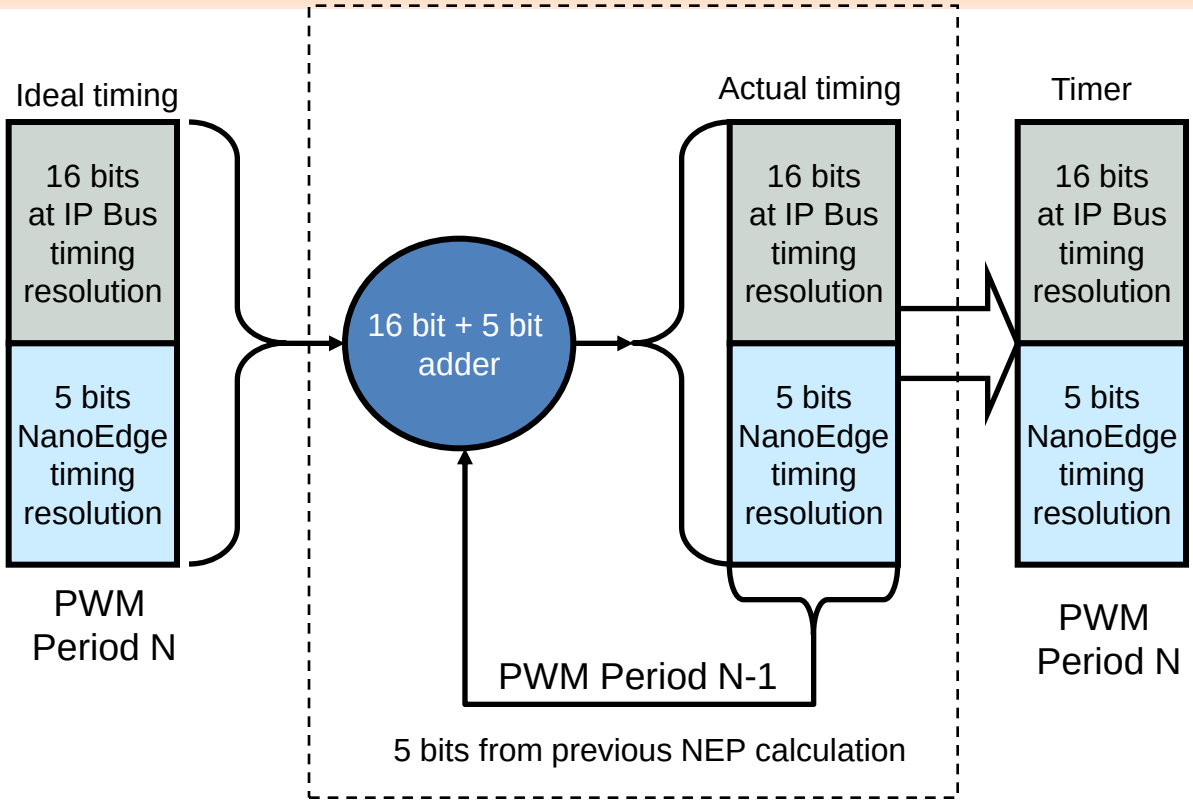
eFlexPWM – High Resolution Frequency Generation

- At high resolution frequency generation
  - Both edges are generated by delay block
  - The analog delay is changing edge by edge every PWM period
  - The analog delay must be calculated every edge or requires some hardware support



eFlexPWM – HR Frequency Generation HW Support

- Need to calculate the next edge position for rising and falling edges within very short period
- Software not fast enough, so need hardware adder
- Diagram shows 21 bit adder to control both edges automatically setting new comparator values after each edge has been triggered



- PWM reload times are restricted to 16-bit IP bus timing (truncation of 21 bit value). Any residual left over from PWM period N-1 needs to be added back to period N.

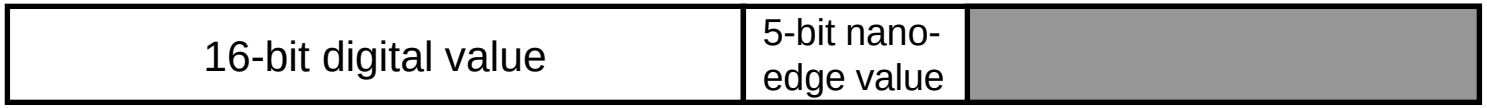
eFlexPWM – HR Frequency Generation

from User Perspective

- The new edge calculation is seem less from user perspective
- The user sets required 21-bit (16+5 bit) values into corresponding value registers only
- The 5-bit delay block corresponds to 3.2 GHz input clock (for 100 MHz PWM module input clock)

PWM Value Register

NanoEdge™ placement not used. 16-bit value only.



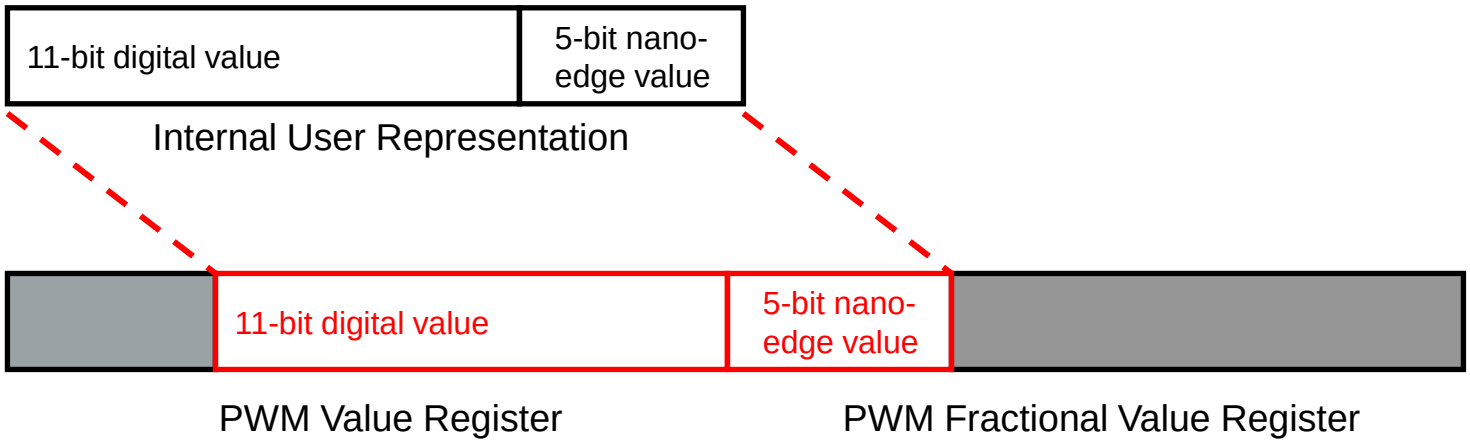
PWM Value Register PWM Fractional Value Register

NanoEdge™ placement enabled. 21-bit (32-bit) value.

eFlexPWM – HR Frequency Generation

from User Perspective

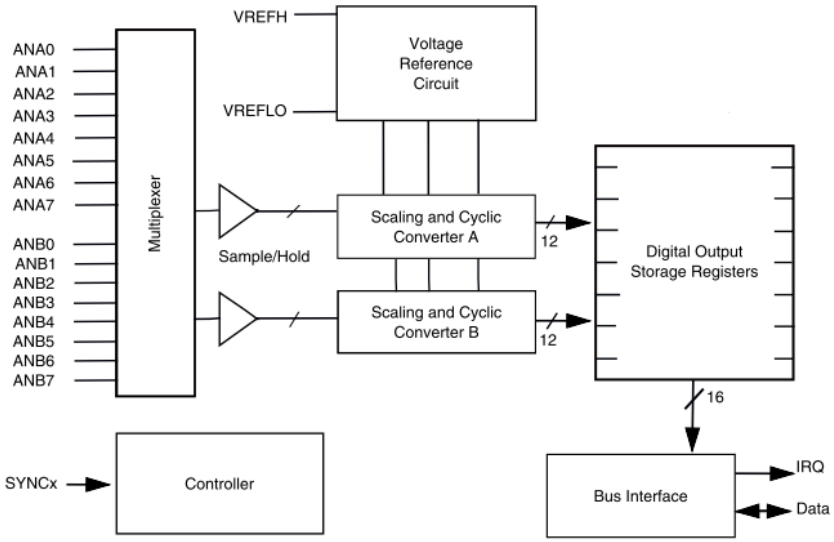
- Example of high resolution calculation
  - For high frequency the digital value is usually less than 11 bits. For example, 200 kHz edge-aligned PWM has resolution less than 9 bits
  - Therefore we can keep calculation in 16 bits
  - Result of calculation is moved right by 5 bits and written into 32-bit register



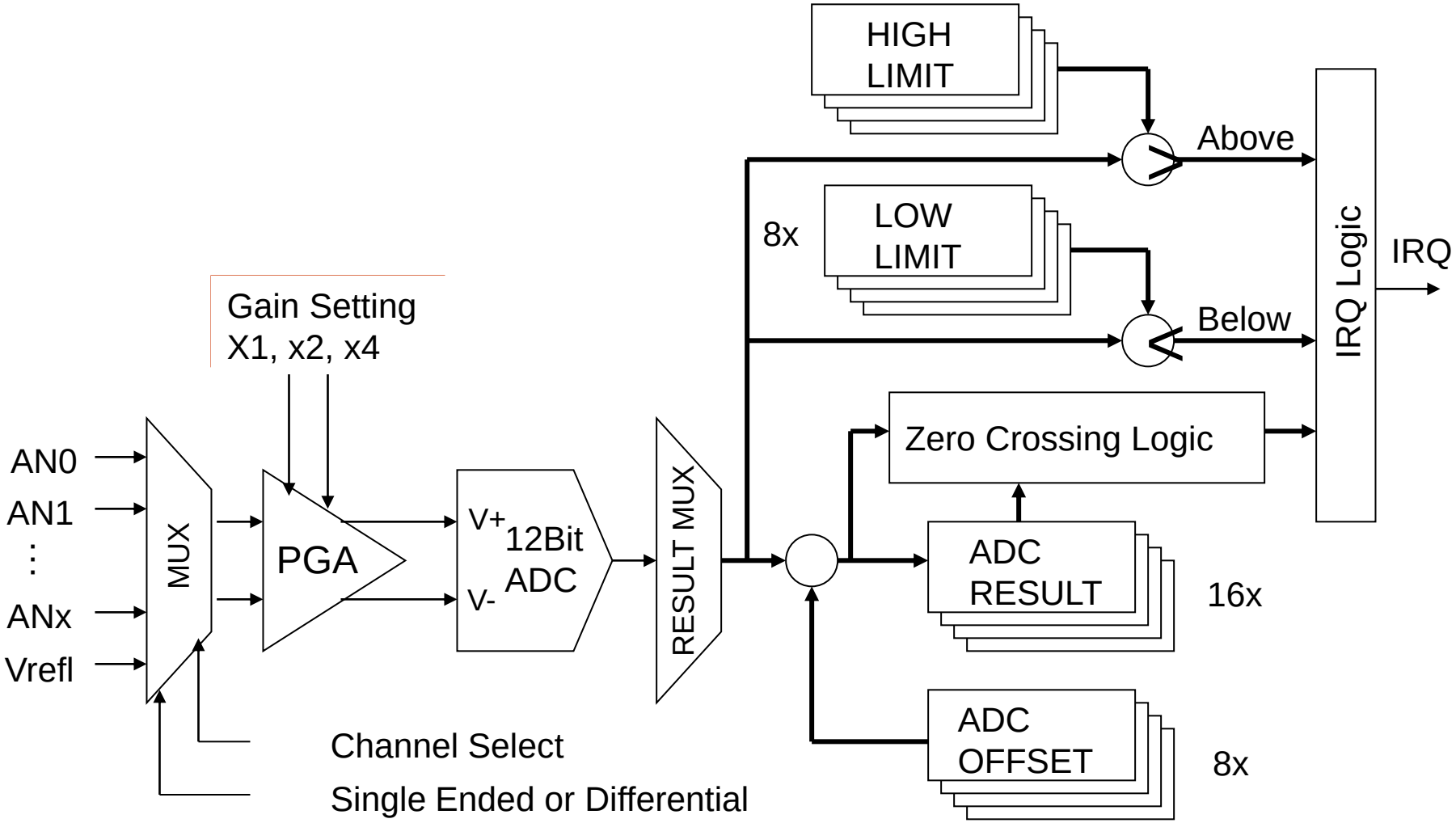
NanoEdge™ placement enabled. 21-bit (32-bit) value.

12-bit Cyclic Analog-to-Digital Converter

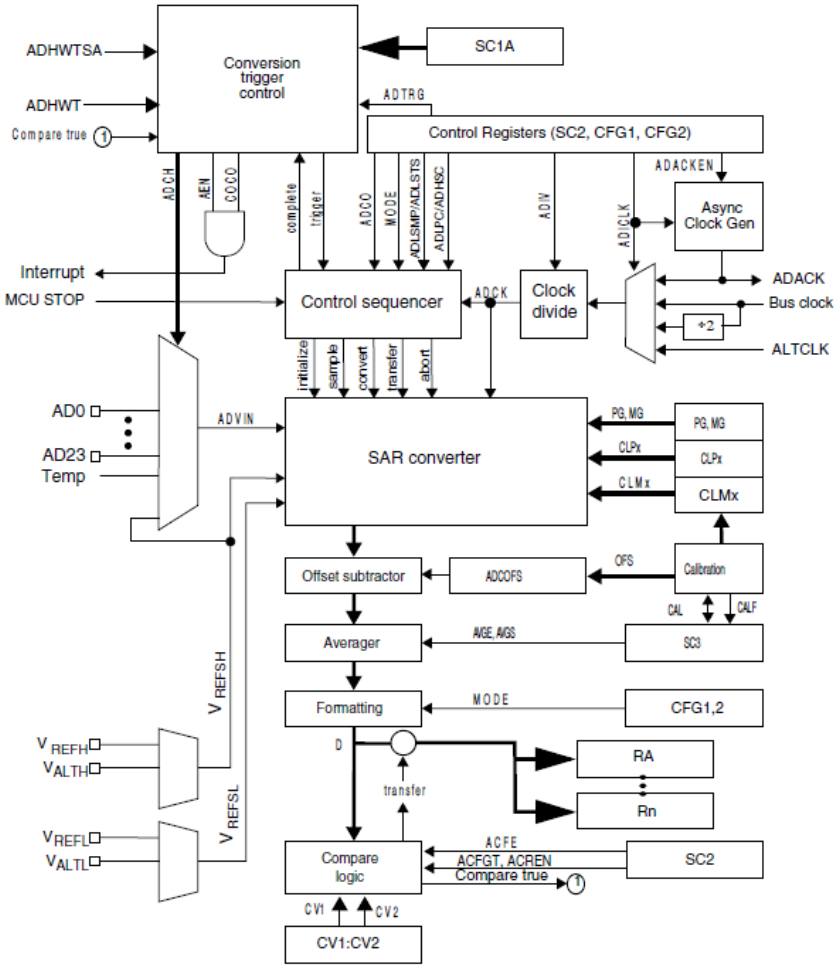
- 12-bit resolution
- Maximum ADC clock frequency of 20 MHz with 50 ns period
- Sampling rate up to 6.66 million samples per second
- Single conversion time of 8.5 ADC clock cycles (8.5 × 50 ns = 450 ns)
- Additional conversion time of 6 ADC clock cycles (6 × 50 ns = 300 ns)
- ADC to PWM synchronization through the SYNC0/1 input signal sequentially scans and stores up to sixteen measurements
- Scans and stores up to eight measurements each on two ADC converters operating simultaneously and in parallel
- Scans and stores up to eight measurements each on two ADC converters operating asynchronously to each other in parallel
- Multi-triggering support
- Gains the input signal by x1, x2, or x4
- Optional interrupts at end of scan if an out-of-range limit is exceeded or there is a zero crossing
- Optional sample correction by subtracting a pre-programmed offset value
- Signed or unsigned result
- Single-ended or differential inputs
- PWM outputs with hysteresis for three of the analog inputs



12-bit Cyclic Analog-to-Digital Converter



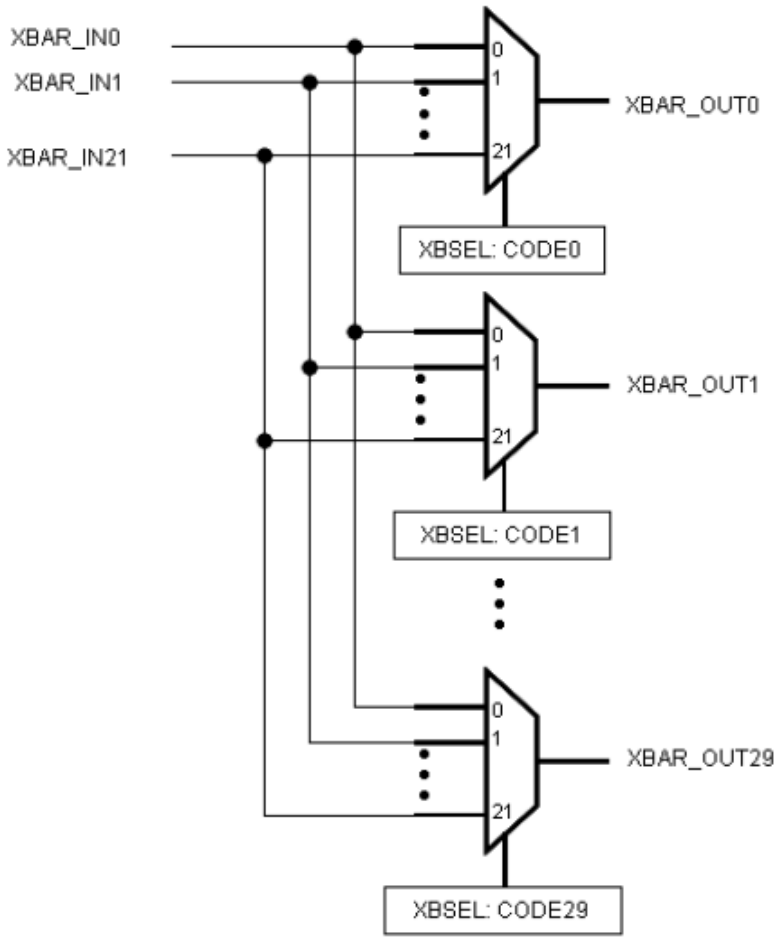
16-bit SAR Analog-to-Digital Converter



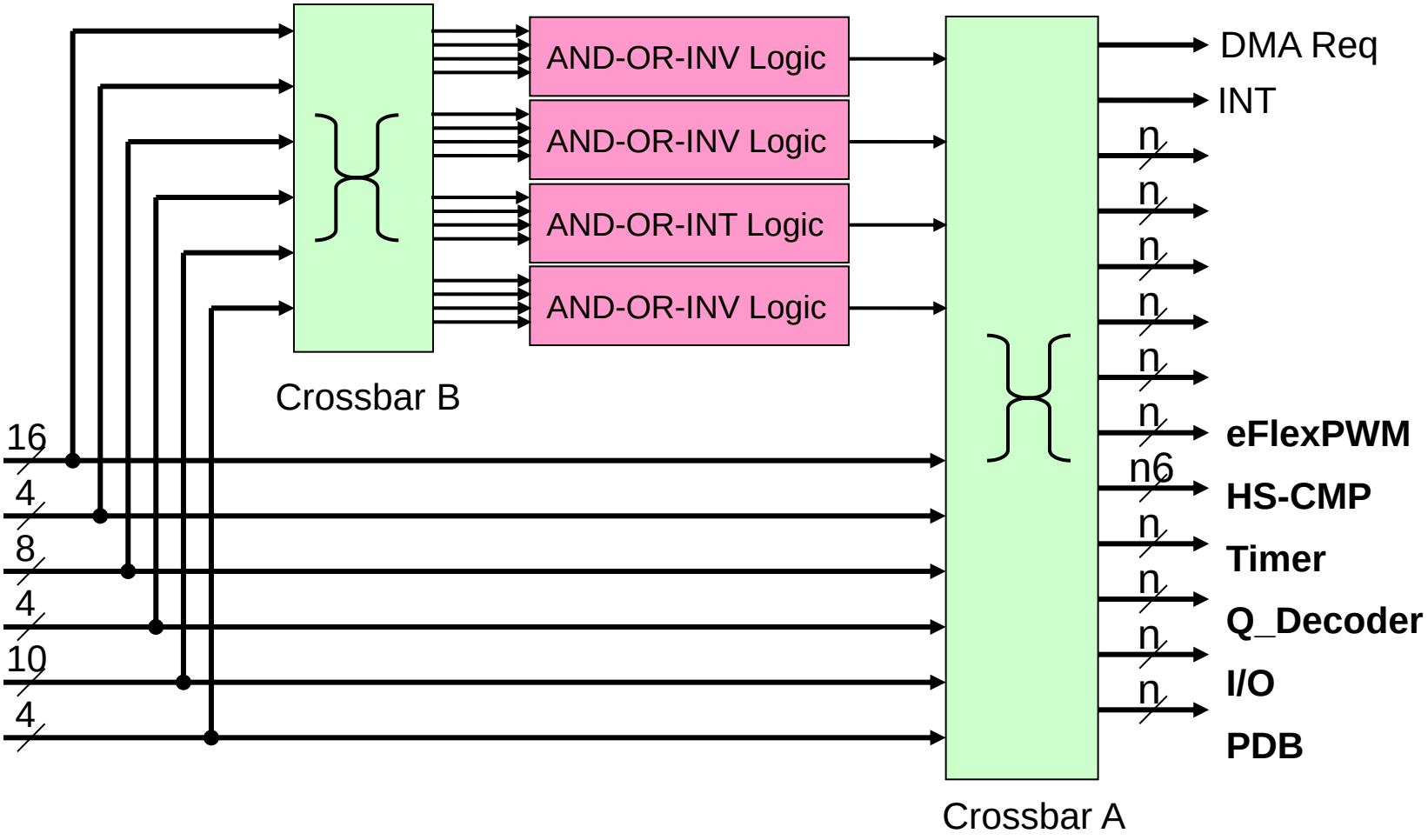
- 24 single-ended external analog inputs
- Single or continuous conversion (automatic return to idle after single conversion)
- Configurable sample time and conversion speed/power
- Input clock selectable from up to four sources
- Operation in low power modes
- Asynchronous clock source for lower noise operation
- Selectable hardware conversion trigger with hardware channel select
- Automatic compare with interrupt for less-than, greater-than or equal-to, within range,or out-of-range, programmable value
- Temperature sensor
- Hardware average function
- Selectable voltage reference: external or alternate
- Self-calibration mode

Inter-Peripheral Crossbar Switch A/B

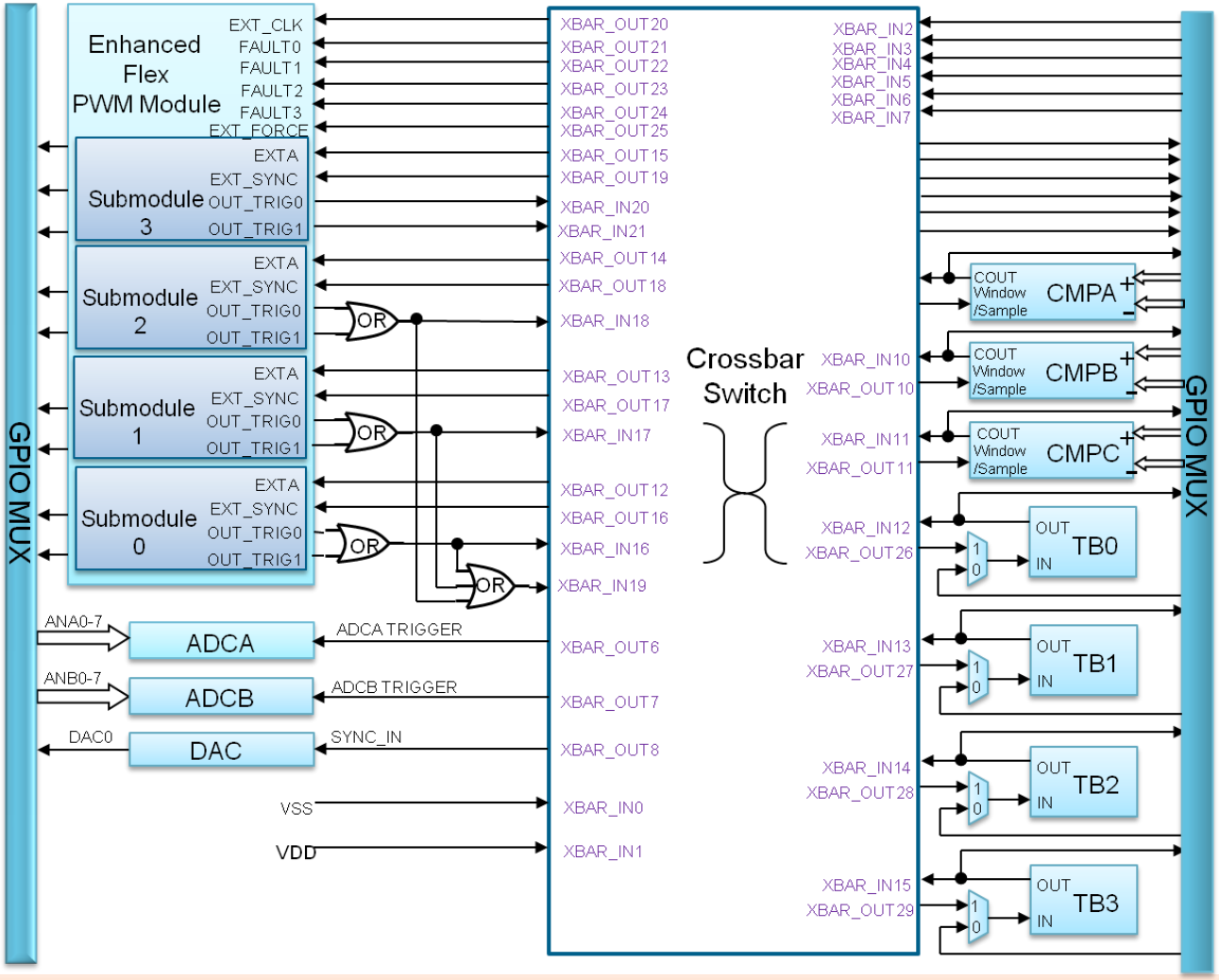
- Flexible signal interconnection among peripherals
- Connects any of 50/26 signals on left side to the output on right side (multiplexer)
- Total 59/16 multiplexers
- All multiplexers share the same set of 50/26 signals
- Increase flexibility of peripheral configuration according to user needs



Inter-Peripheral Crossbar Switch A/B + AOI Logic



Inter-Peripheral Crossbar Switch - MC56F824x/5x



CALLING CONVENTIONS

Calling Conventions & Stack Frames

- Calling conventions
  - Language dependent
  - Processor dependent
    - Number of registers available for storing the parameters
    - Efficiency of data storage – byte, word (16-bit) , long (32-bit)
- Passing values to functions
  - Compiler scans the list of parameters from left to right
  - Compiler uses the registers of a processor to pass the values to functions
  - Compiler passes remaining parameter values on the stack
    - Stack has to be handled properly in terms of alignment, incrementing and decrementing
- Returning values from functions
  - Compiler returns function results in registers, stack or …..

Calling Conventions – DSC56F8025

- Passing values to functions – 56800E core
  - Compiler uses the following registers to pass parameter values to functions – A, B, R1, R2, R3, R4, Y0 and Y1
  - Upon a function call, the compiler scans the list of parameters from left to right and uses the registers as follows:
    - The first two 8-bit or 16-bit integer/fractional values – Y0 and Y1
    - The first two 32-bit integer/fractional values – A and B
    - The first four pointer parameter values – R2, R3, R4 and R1 (in that order)
    - The third and fourth 8-bit or 16-bit integer/fractional value – A and B (provided that the compiler does not use these registers for 32-bit parameter values)
    - The third 8-bit or 16-bit integer/fractional value – B (provided that the compiler does not use these registers for 32-bit parameter value)
    - The remaining parameters – compiler passes on the stack
      - The system increments the stack by the total amount of space required for memory parameters
      - This incrementing must be an even number of words, as the stack pointer (SP) must be continuously long-aligned
      - The system moves parameter values to the stack from left to right, beginning with
      - the stack location closest to the SP
      - Because a long parameter must begin at an even address, the compiler introduces one-word gaps before long parameter values, as appropriate

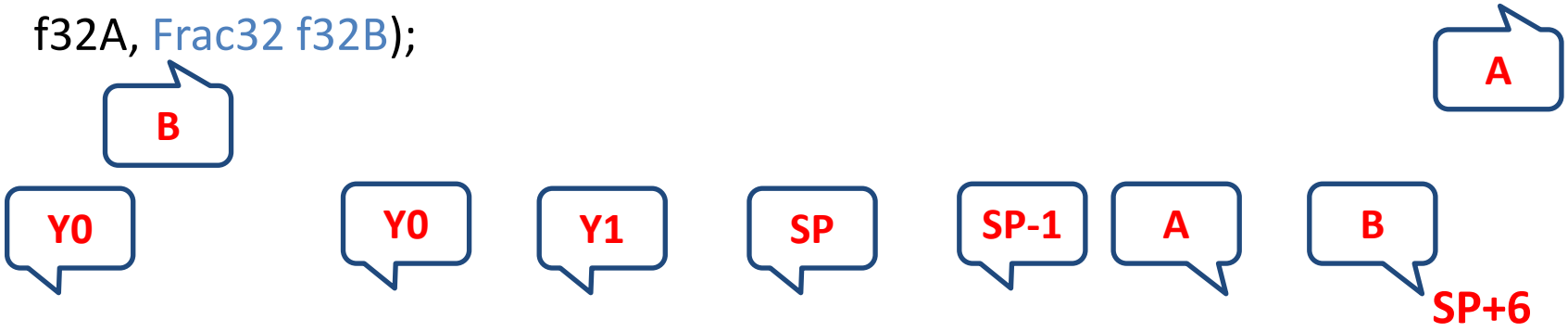
Calling Conventions – DSC56F8025 cont'd

- Returning values from functions
  - Compiler returns function results in registers
    - 8-bit integer values — Y0.
    - 16-bit integer values — Y0.
    - 32-bit integer or float values — A.
    - All pointer values — R2.
    - Structure results — R0 contains a pointer to a temporary space allocated by the caller. (The pointer is a hidden parameter value.)
- Volatile and non-volatile registers
  - Non-Volatile registers – C0, C1, C10, D0, D1, D10 and R5 (in some cases)
    - Values in volatile registers can be saved across functions calls
    - Another term for such registers – SOC (Saved Over a Call) registers
  - Volatile registers – all ALU, AGU except Non-Volatile registers
    - Values in volatile registers cannot be saved across functions calls
    - Another term for such registers – non-SOC registers

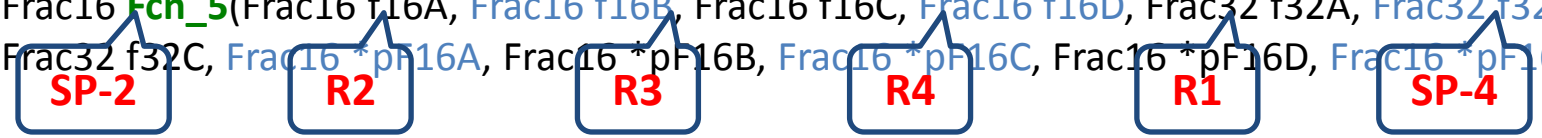
Calling Convention - Example

- Frac16 Fcn_4(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D);

- Frac16 Fcn_3(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D, Frac32 f32A, Frac32 f32B);



- Frac16 **Fcn_5**(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D, Frac32 f32A, Frac32 f32B, Frac32 f32C, Frac16 *pF16A, Frac16 *pF16B, Frac16 *pF16C, Frac16 *pF16D, Frac16 *pF16E);



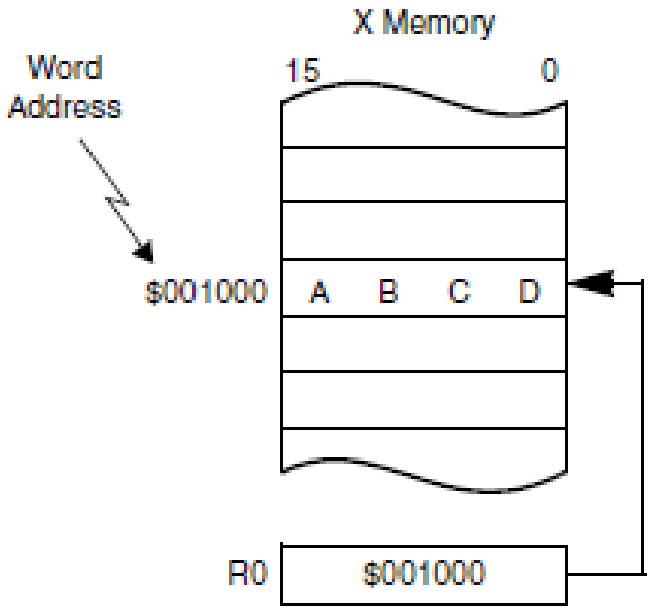
STACK FRAME & MEMORY/STACK ALIGNMENT

Memory Access and Pointers – Data Alignment

- The DSP56800E core was designed to operate as a word-addressable machine
- Instruction set allows to access: byte, word and long-word
- Accessing word values using word pointers

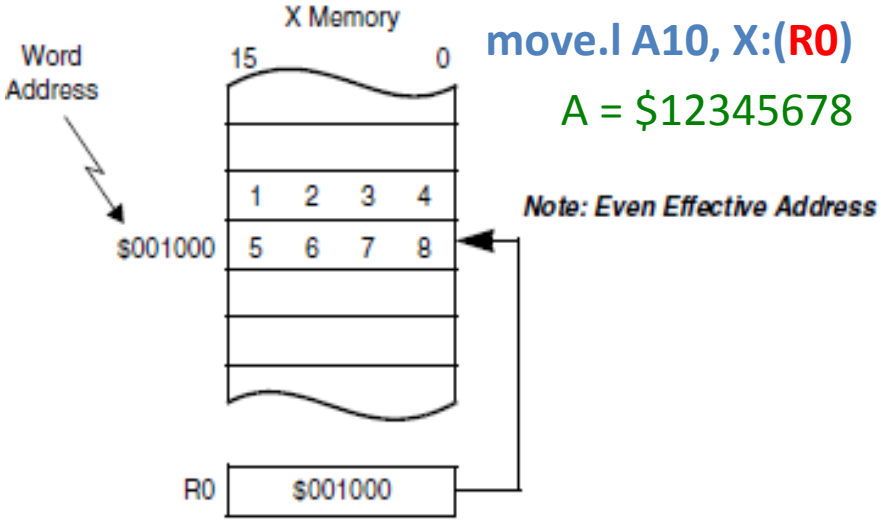
move.w A1, X:(R0)

A1 = $ABCD

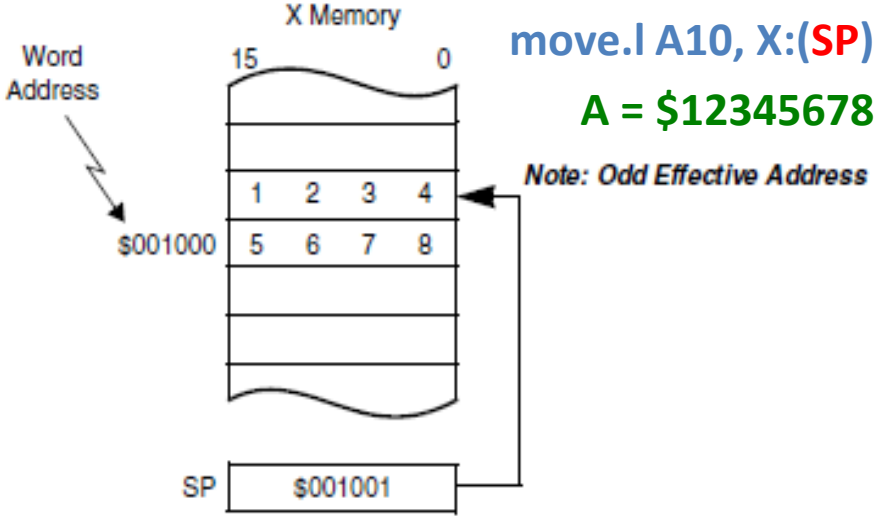


Memory Access and Pointers – Data Alignment cont'd

- Accessing a long word in memory – always word address
- Always aligned to even word address
- Even address holds the lower word
- Odd address holds the upper word

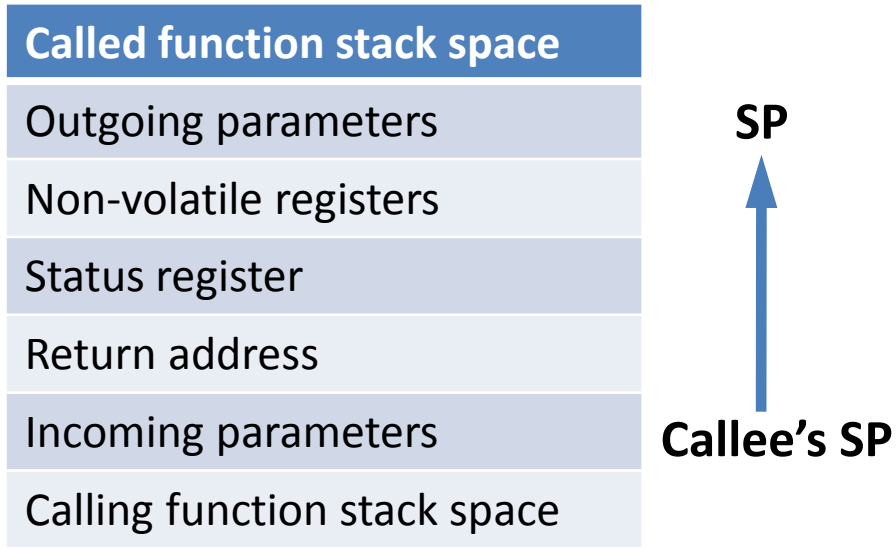


- Accessing a long word on stack – always word address
- Always aligned to even word address
- Even address holds the lower word
- Odd address holds the upper word



Stack Frame and Alignment

- Compiler always must operate with the SP long aligned!!!!!!!
- Start-up code in runtime firs initializes the SP to an odd value
- At all times after, the SP must point to an odd word address
- The compiler never generates instruction that adds or subtracts an odd value from SP
- The compiler never generates:
  - move.w or moveu.w with X:(SP)+ or X:(SP)-

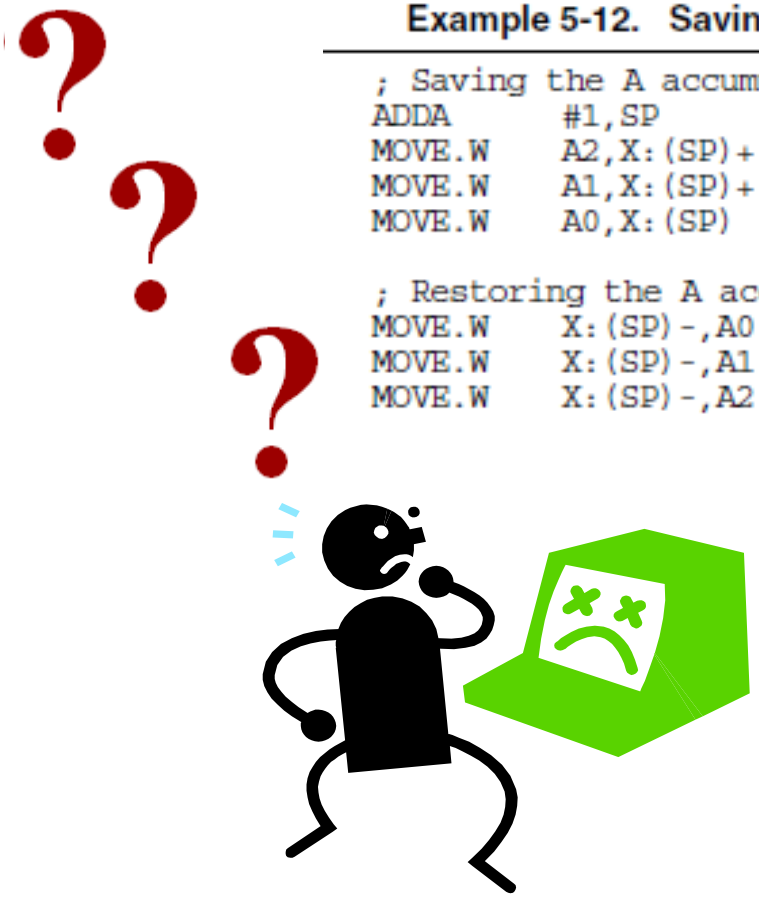


Example from the DSC Manual

Example 5-12. Saving and Restoring an Accumulator—Word Accesses

```
; Saving the A accumulator to the stack
ADDA      #1,SP          ; Point to first empty location
MOVE.W    A2,X:(SP)+     ; Save extension register
MOVE.W    A1,X:(SP)+     ; Save A1 register
MOVE.W    A0,X:(SP)      ; Save A0 register

; Restoring the A accumulator from the stack
MOVE.W    X:(SP)-,A0     ; Restore A0 register
MOVE.W    X:(SP)-,A1     ; Restore A1 register
MOVE.W    X:(SP)-,A2     ; Restore extension register
```



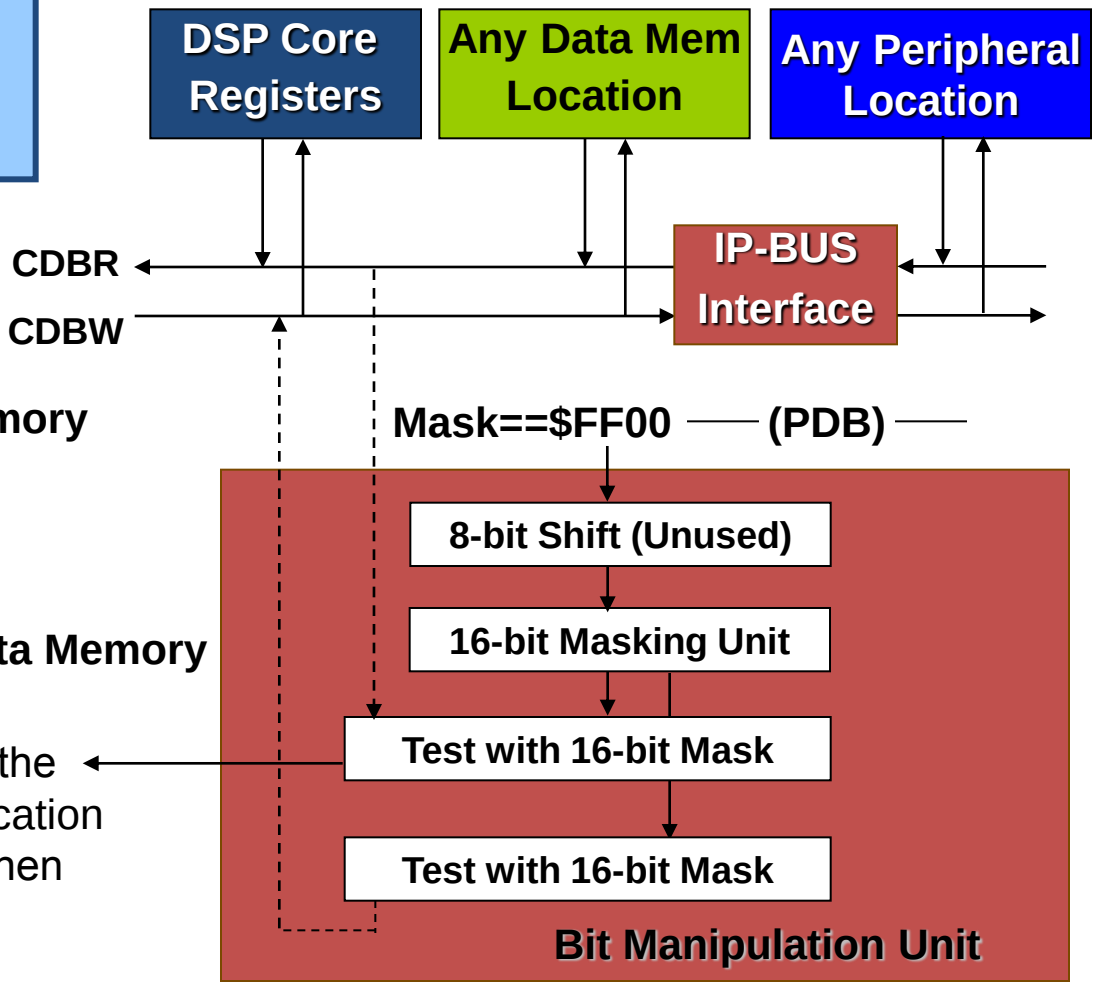
Risk of misaligned data access

REGISTER BIT MANIPULATION & BIT MANIPULATION UNIT

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;

REGISTER BIT MANIPULATION

Register Bit Manipulation

- What needs to take into consideration?
  - Addressing capability of microcontroller
    - Byte, Word, 32-bit
    - Bit addressing capability
  - Register content
    - Control bits only – single control bits
    - Control bits only – multi control bits
    - Flag bits only - asynchronous
    - Control bits and one flag bit
    - Control bits and flag bits
  - Microcontroller support
    - Bit manipulation using bit addressing capability
    - Bit manipulation using ALU
    - Bit manipulation using special bit manipulation instructions
  - Compiler support
    - C language statement
    - Specific macro forcing the compiler to generate correct source code
    - Assembler

Register Bit Manipulation – Single Bits

Address 0x0F000 - Timer Control Register (TMR_CTRL)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CM				PCS				SCS				ONCE	LENGTH	DIR	Co_INIT	OM			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Control bits only – single and multiple bits
  - Single bits – ONCE, LENGTH, DIR, Co_INIT
  - Multiple bit fields - CM,PCS, SCS, OM

Register Bit Manipulation – Single Bits cont'd

Address 0x0F000 - Timer Control Register

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CM				PCS				SCS		ONCE	LENGTH	DIR	Co_INIT	OM	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Single bit - DIR

```
#define TMR_CTRL_DIR        0x0010  /* DIR bit mask */
#define TMR_CTRL_ADDR       0xF000  /* Addr of TMR_CTRL */
```

C-language:

```
(*(UWord16*)TMR_CTRL_ADDR) |= TMR_CTRL_DIR; /* Set DIR bit */
(*(UWord16*)TMR_CTRL_ADDR) &= ~TMR_CTRL_DIR; /* Clear DIR bit */
```

Register Bit Manipulation – Single Bits cont'd

Single bit – DIR

Compiler result - the most optimal solution:

56F800E Assembler – support of bit manipulation instructions:

```
bfset #TMR_CTRL_DIR,X:TMR_CTRL_ADDR /*Set DIR bit*/
bfclr #TMR_CTRL_DIR,X:TMR_CTRL_ADDR /*Clear DIR bit*/
```

Compiler result:

56F800E Assembler – support of bit manipulation instructions:

```
move.w #TMR_CTRL_ADDR,R0
bfset  #TMR_CTRL_DIR,X:(R0);/* Set DIR bit */
move.w #TMR_CTRL_ADDR,R0
bfclr  #TMR_CTRL_DIR,X:(R0);/* Clear DIR bit */
```

Register Bit Manipulation – Single Bits cont'd

Single bit – DIR

Compiler result – without support of bit manipulation instructions

56F800E Assembler:

```
move.w X:TMR_CTRL_ADDR,Y0
move.w #TMR_CTRL_DIR,X0
or.w   X0,Y0
move.w Y0,X:TMR_CTRL_ADDR
```

56F800E Assembler:

```
move.w X:TMR_CTRL_ADDR,Y0
move.w #TMR_CTRL_DIR,X0
not.w  X0
and.w  X0,Y0
move.w Y0,X:TMR_CTRL_ADDR
```

Register Bit Manipulation – Multiple Bit Fields

Address 0x0F000 - Timer Control Register

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CM				PCS				SCS		ONCE	LENGTH	DIR	Co_INIT	OM	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Multiple bit fields – SCS

- Difficult to configure two bits (SCS) using single instruction
- Case 1:
  - Initial SCS value = 01
  - Target SCS value = 10
- Using bfclr and bfset instructions is not correct
- ALU support or special bit manipulation instruction is needed

Register Bit Manipulation – Multiple Bit Fields cont'd

Address 0x0F000 - Timer Control Register

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	CM				PCS				SCS				ONCE	LENGTH	DIR	Co_INIT	OM			
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Multiple bit fields – SCS

```
#define TMR_CTRL_SCS      0x0180  /* SCS bit mask */
#define TMR_CTRL_ADDR     0xF000  /* Addr of TMR_CTRL */
#define TMR_CTRL_SCS_SET  0x0100  /* SCS configuration */
```

C-language:

```
(*(UWord16*)TMR_CTRL_ADDR) &= ~TMR_CTRL_SCS;/* Clear SCS bits */
(*(UWord16*)TMR_CTRL_ADDR) |= TMR_CTRL_SCS_SET;/*Set SCS bits */
```

Register Bit Manipulation – Multiple Bit Fields cont'd

Multiple bit fields – SCS

Compiler result – ALU support

56F800E Assembler:

```
move.w #TMR_CTRL_SCS,X0
not.w  X0
move.w X:TMR_CTRL_ADDR,Y0
and.w  X0,Y0
move.w #TMR_CTRL_SCS_SET,X0
or.w   X0,Y0
move.w Y0,X:TMR_CTRL_ADDR
```

Register Bit Manipulation – Multiple Bit Fields cont'd

Multiple bit fields – SCS

56F800EX core support:

56F800EX Assembler – support of bit manipulation instruction:

Test Bitfield and Set/Clear instruction

- Single atomic non-interruptible operation
- 3 clocks instruction
- Universal approach for manipulation with any bit or set of bits

```
bfsc #TMR_CTRL_SCS,TMR_CTRL_SCS_SET,X:TMR_CTRL_ADDR
```

BFSC Macro Instruction

BFSC — Test Bitfield and Set/Clear — BFSC

Operation:

0/1 → (<bitfield> of destination) (no parallel move)

Assembler Syntax:

BFSC #iiii,#iiii,X:<ea>(no parallel move)

BFSC bit_select_mask,operation_mask,X:<ea>

Description: Test all selected bits of the destination operation. The operand size is always word. The first #<MASK16> field is the bit select mask. It is used to specify which bits are tested and then set or cleared. The second #<MASK16> field is the operation mask. It is used to specify if a bit selected for testing is set or cleared. The bits that are set in the immediate value of the bit select mask are the same bits that are tested and set or cleared in the destination; the bits that are cleared in the immediate value of the bit select mask are ignored in the destination. If a bit in the operation mask corresponding to a selected bit in the immediate value of the bit select mask is set, then the corresponding bit in the destination is set. If a bit in the operation mask corresponding to a selected bit in the immediate value of the bit select mask is cleared, then the corresponding bit in the destination is cleared.

This instruction performs a read-modify-write operation on the destination memory location and requires two destination accesses. Operation:

```
"C" condition code bit = ((source[15:0] & bit_select_mask[15:0]) ==
(operation_mask[15:0] & bit_select_mask[15:0]));
destination[15:0] = ((source[15:0] & ~bit_select_mask[15:0]) |
(operation_mask[15:0] &  bit_select_mask[15:0]));
```

Usage: This instruction is very useful in performing I/O and flag bit manipulation.

Instruction Fields:

Operation	Operands	C	W	Comments
BFSC	#<MASK16>,#<MASK16>,X:(Rn)	3	3	Test bitfield and set/clear

Register Bit Manipulation – Control Bits and Single Flag

Address 0x0F060 - PIT Control Register (PIT_CTRL)

Address: Base address + 0h offset

Bit	15	14	13	12	11	10	9	8
Read	SLAVE	0					CLKSEL	
Write								
Reset	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
Read	0	PRESCALER				PRF	PRIE	CNT_EN
Write								
Reset	0	0	0	0	0	0	0	0

PRF flag is cleared by writing one, writing zero has no effect

- Control bits – single and multi bits
  - Single bits – PRIE, CNT_EN
  - Multiple bit fileds - SLAVE, CLKSEL, PRESCALER
- Flag
  - Single flag – PRF(Pit Roll-Over Flag)

Register Bit Manipulation – Control Bits and Single Flag cont'd

Address 0x0F060 - PIT Control Register (PIT_CTRL)

Address: Base address + 0h offset

Bit	15	14	13	12	11	10	9	8
Read	SLAVE	0					CLKSEL	
Write								
Reset	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
Read	0	PRESCALER				PRF	PRIE	CNT_EN
Write								
Reset	0	0	0	0	0	0	0	0

```
#define PIT_CTRL_PRF      0x0004  /* PRF bit mask */
#define PIT_CTRL_ADDR     0xF060  /* Addr of PIT_CTRL */
```

C-language:

```
(*(UWord16*)PIT_CTRL_ADDR) |= PIT_CTRL_PRF;/* Write one to PRF bit */
```

Assembler:

```
bfset #PIT_CTRL_PRF, X:PIT_CTRL_ADDR) /* Write one to PRF bit */
```

Register Bit Manipulation – Control Bits and Single Flag cont'd

Address 0x0F060 - PIT Control Register (PIT_CTRL)

Address: Base address + 0h offset

Bit	15	14	13	12	11	10	9	8
Read	SLAVE	0					CLKSEL	
Write								
Reset	0	0	0	0	0	0	0	0

Bit	7	6	5	4	3	2	1	0
Read	0	PRESCALER				PRF	PRIE	CNT_EN
Write								
Reset	0	0	0	0	0	0	0	0

```
#define PIT_CTRL_ADDR        0xF060 /* Addr of PIT_CTRL */
```

Another approach to clear PRF flag by writing one

Warning!!! Compiler usually does not compile the following line of the code. It recognized as the code doing nothing

C-language:

```
(*(UWord16*)PIT_CTRL_ADDR) &= 0xFFFF;/* Write one to PRF bit */
```

Assembler:

```
bfclr #0x0000,X:PIT_CTRL_ADDR) /* Write one to PRF bit */
```

Register Bit Manipulation – Control Bits and Single Flag cont'd

Control bits & single flag

Without support of bit manipulation instructions, ALU is used

56F800E Assembler:

```
move.w X:PIT_CTRL_ADDR,Y0
move.w #PIT_CTRL_PRF,X0
or.w   X0,Y0
move.w Y0,X:PIT_CTRL_ADDR
```

56F800E Assembler:

```
move.w X:PIT_CTRL_ADDR,Y0
move.w #0xffff,X0
and.w  X0,Y0
move.w Y0,X:TMR_CTRL_ADDR
```

Low-level Drivers

- **Why not to use direct access to peripheral registers?**
  - Most of **`ioctl`** calls are “macroized” to direct register access anyway (either read/write or bit-set/bit-clear instructions used)
  - Some registers do need special attention, **`ioctl`** usage brings kind-of **abstraction** and **transparency** to an application code while still being optimally compiled

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

Exercise: Suppose you want to clear DIRQ bit only, while not modifying the rest of the register. Also you must not clear the HIRQ and XIRQ bits.
What C or assembly statement will you use on 56F800E? solution on the next slide...

Low-level Drivers: Exercise

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

```
#define DECCR_DIRQ 0x0010   /* DIRQ bit constant */
ArchIO.Decoder0.deccr       /* register in the peripheral structure */
```

C-language:

```
    ArchIO.Decoder0.deccr = DECCR_DIRQ;
```

56F800E Assembler:

```
    asm ( move.w #>16,X:0x00f180 );
```

- DIRQ gets cleared ... OK
- XIRQ and HIRQ remain unchanged ... OK
- All other bits get reset! ... Wrong!

Low-level Drivers: Exercise

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

```
#define DECCR_DIRQ 0x0010   /* DIRQ bit constant */
ArchIO.Decoder0.deccr       /* register in the peripheral structure */
```

C-language:

```
    ArchIO.Decoder0.deccr |= DECCR_DIRQ;
```

56F800E Assembler:

```
    asm ( bfset #0x10,X:0x00f180 );
```

- DIRQ gets cleared ... OK
- Other register bits unchanged ... OK
- XIRQ or HIRQ gets reset if they read as “1” (i.e. when interrupt request is pending!)

Low-level Drivers: Exercise

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

```
#define DECCR_DIRQ 0x0010      /* DIRQ bit constant */
#define DECCR_HIRQ 0x8000      /* HIRQ bit constant */
#define DECCR_XIRQ 0x0100      /* XIRQ bit constant */
ArchIO.Decoder0.deccr          /* register in the peripheral structure */
```

C-language:

```
ArchIO.Decoder0.deccr &= ~(~(DECCR_DIRQ) &
    (DECCR_HIRQ | DECCR_XIRQ));
```

56F800E Assembler:

```
asm ( bfclr #0x8100,X:0x00f180 );
```

Low-level Drivers: Exercise

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

```
#define DECCR_DIRQ 0x0010      /* DIRQ bit constant */
#define DECCR_HIRQ 0x8000      /* HIRQ bit constant */
#define DECCR_XIRQ 0x0100      /* XIRQ bit constant */
ArchIO.Decoder0.deccr          /* register in the peripheral structure */

C-language:
     ArchIO.Decoder0.deccr &= ~(~(DECCR_DIRQ) &
56F8
```

Better work with Quick_Start and use the "Clear Interrupt Request" command:

```
ioctl(DEC_0, DEC_INT_REQUEST_CLEAR, DEC_DECCR_DIRQ);
```

Register Bit Manipulation – Control Bits and Multiple Flags

Decoder Control Register (DECCR)

Base + $0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	HIRQ	HIE	HIP	HNE	0	REV	PH1	XIRQ	XIE	XIP	XNE	DIRQ	DIE	WDE	MODE	
Write					SWIP											
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Clear-by-write-one interrupt request flags

```
#define DECCR_DIRQ 0x0010          /* DIRQ bit constant */
#define DECCR_HIRQ 0x8000          /* HIRQ bit constant */
#define DECCR_XIRQ 0x0100          /* XIRQ bit constant */
#define DECCR_DIRQ_HIRQ_XIRQ (DECCR_DIRQ | DECCR_HIRQ | DECCR_XIRQ)
ArchIO.Decoder0.deccr              /* register in the peripheral structure */

C-language:
     ??????

56F800E Assembler:
     asm (bfsc # DECCR_DIRQ_HIRQ_XIRQ, DECCR_DIRQ,X:0x00f180);
```

FRACTIONAL VS. INTEGER ARITHMETIC

Fractional vs. Integer Data Types

Fractional data types

- Signed Fractional

$$-1 \le SF \le +1.0 - 2^{-[N-1]}$$

- The most negative number
  - Word: -1.0
  - Long word: -1.0
- Internal representation of the most negative value
  - Word: 0x8000
  - Long word: 0x80000000

Integer data types

- Signed Integer

$$-2^{[N-1]} \le SI \le [2^{[N-1]} - 1]$$

- The most negative number
  - Word: -32768
  - Long Word: -2,147,483,648
- Internal representation of the most negative value
  - Word: 0x8000
  - Long word: 0x80000000

Fractional vs. Integer Data Types cont'd

Fractional data types

- Signed Fractional

$$-1 \le SF \le +1.0 - 2^{-[N-1]}$$

- The most positive number
  - Word: $1.0 - 2^{-15} = 0.999969\ldots$
  - Long word: $1.0 - 2^{-31} = 0.999999\ldots$
- Internal representation of the most negative value
  - Word: 0x7FFF
  - Long word: 0x7FFFFFFF

Integer data types

- Signed Integer

$$-2^{[N-1]} \le SI \le [2^{[N-1]} - 1]$$

- The most negative number
  - Word: -32768
  - Long Word: -2,147,483,648
- Internal representation of the most negative value
  - Word: 0x7FFF
  - Long word: 0x7FFFFFFF

Float vs. Double Data Types – IEEE 754 Standard

Floating point number: 32-bit

- S – Sign bit
- E – Exponent bits
- F – Fraction bits

- The number is represented from the left to right

- Sign bit – 1
- Exponent bits - 8
- Fraction bits - 23

Double floating point number: 64-bit

- S – Sign bit
- E – Exponent bits
- F – Fraction bits

- The number is represented from the left to right

- Sign bit – 1
- Exponent bits - 11
- Fraction bits - 52

Fractional to Integer / Integer to Fractional

16-bit Fractional Value = Integer Value / (2^{15})

32-bit Fractional Value = Integer Value / (2^{31})

16-bit Integer Value = Fractional Value * 2^{15}

32-bit Integer Value = Fractional Value / * 2^{31}

Fractional vs. Integer Arithmetic

	Binary	Hexadecimal	Integer	Fractional
1	0111	0x7	7	0.875
2	0110	0x6	6	0.750
3	0101	0x5	5	0.625
4	0100	0x4	4	0.500
5	0011	0x3	3	0.375
6	0010	0x2	2	0.250
7	0001	0x1	1	0.125
8	0000	0x0	0	0
9	1111	0xF	-1	-0.125
10	1110	0xE	-2	-0.250
11	1101	0xD	-3	-0.375
12	1100	0xC	-4	-0.500
13	1011	0xB	-5	-0.625
14	1010	0xA	-6	-0.750
15	1001	0x9	-7	-0.875
16	1000	0x8	-8	-1.000

- Example: 4-bit word
- N = 4
- Number of numbers created from 4-bit word: $2^4 = 16$
- Range of fractional values

$$-1 \le SF \le +1.0 - 2^{-[N-1]}$$

$$-1 \le SF \le +0.875$$

- Range of integer values

$$-2^{[N-1]} \le SI \le [2^{[N-1]} - 1]$$

$$-8 \le SI \le 7$$

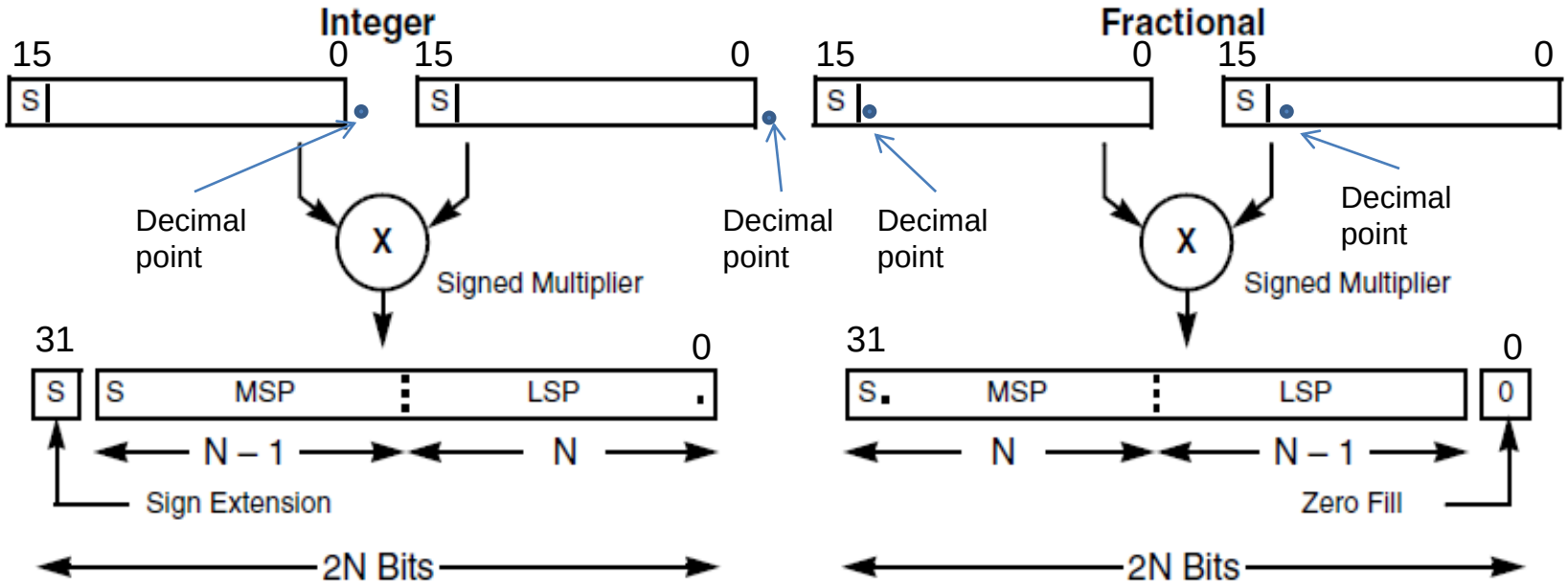
Fractional vs. Integer Arithmetic

- Example 1: addition of two numbers
  - Assumption – 4-bit word

    Binary representation 0100 + 0011 = 0111
  - A) Fractional mathematic
    - 0100 = 0.500 and 0011 = 0.375
    - 0.5 00 + 0.375 = 0.875 binary representation of 0.875 = 0111
  - B) Integer mathematic
    - 0100 = 4 and 0011 = 3
    - 4+ 3 = 7 binary representation of 7 = 0111

  - Relation between integer and fractional result
    - Integer = Fractional * 2^3 => Integer = 0.875 * 2^3 = 7
    - Fractional = Integer / 2^3 => Fractional = 7 / 2^3 = 0.875
  - No difference between fractional and integer addition
  - No difference between fractional and integer subtraction

Fractional and Integer Multiplication

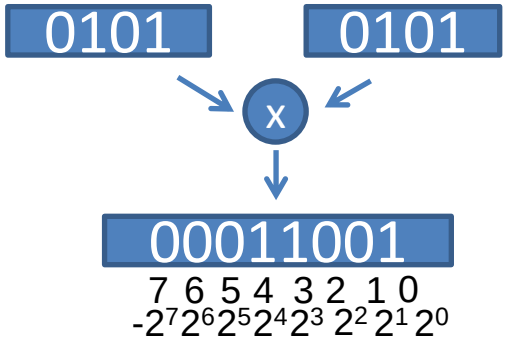


Fractional and Integer Multiplication

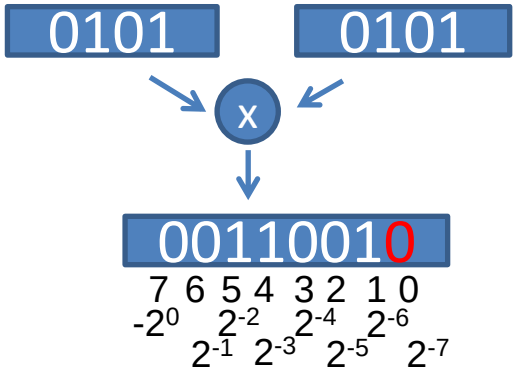
- Example 2: multiplication of two numbers
  - Assumption – 4-bit word

Binary representation 0101 * 0101

- Integer multiplication



Fractional multiplication



- 0101 * 0101 => 5 * 5 = 25 => 0.625 * 0.625 = 0.390625
- $-2^7*0+2^6*0+2^5*0+2^4*1+2^3*1+2^2*0+2^1*0+2^0*1 = 25$
- $-2^0*0+2^{-1}*0+2^{-2}*1+2^{-3}*1+2^{-4}*0+2^{-5}*0+2^{-6}*1+2^{-7}*0 = 0.390625$
- ??????????????!!! $0.390625 * 2^7 = 50$!!!??????????????

Fractional and Integer Multiplication cont'd

- The multiplication operation is not the same for integer and fractional arithmetic
- The result of a fractional multiplication differs from the result of an integer multiplication. The difference amounts to a 1-bit shift of the final result

Result of fractional multiplication is twice bigger than result of integer multiplication

FRACTIONAL ARITHMETIC USE & APPLICATION SCALING

Fractional Arithmetic Use

- V = R * I
- Scaling to the fractional arithmetic
  - V – variable with maximum value equal to V_m
  - I - variable with maximum value equal to I_m
  - R - constant

$$\frac{V}{V_m I_m} = R \cdot \frac{I}{V_m I_m} \qquad \frac{V}{V_m} = \frac{I_m}{V_m} \cdot R \cdot \frac{I}{I_m}$$

  - Equation scaled into the fractional arithmetic

$$V' = R' \cdot I'$$

  - Where $V' = \frac{V}{V_m}$ $I' = \frac{I}{I_m}$ $R' = R \cdot \frac{I_m}{V_m}$
  - V' ,I' ,R' are the dimensionless quantity (without physical dimension)

Analogue Quantities Scaling

- Analogue quantities (voltage, current, frequency) are scaled to the maximum measurable range – depended on hardware
- Relation between a real and a fractional representation

$$\text{Fractional Value} = \frac{\text{Real value}}{\text{Real quantity Range}}$$

  - Fractional Value – fractional representation of the real value [Frac16]
  - Real Value – real value of the quantity [V, A, RPM, etc.]
  - Real Quantity Range – maximum range of the quantity, defined in the application [V,A,RPM, etc.]

- Angles are represented as a 16-bit fractional values in the range [-1,1) which corresponds to the angle [-PI,PI)

$$-\text{pi} \approx 0\text{x}8000$$

$$\text{pi} \bullet (1.0 - 2^{-15}) \approx 0\text{x}7\text{FFF}$$

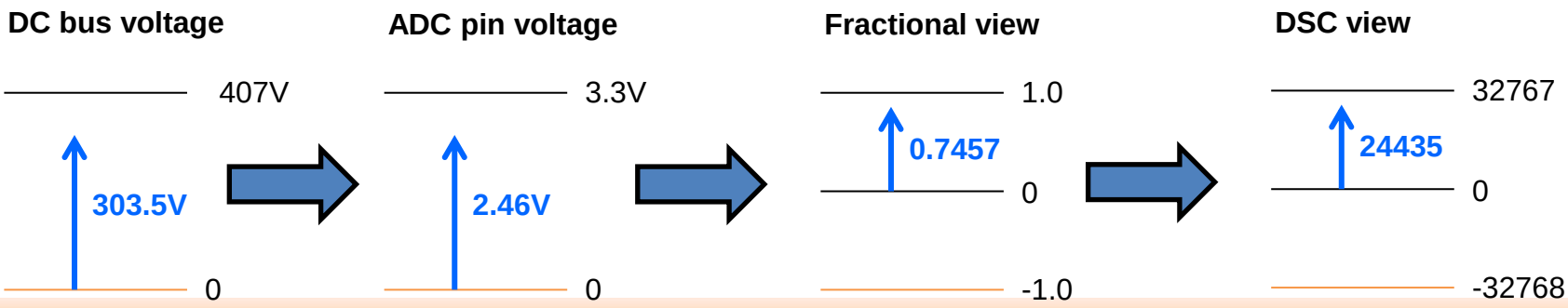
Analogue Quantities Scaling

- Example:
  - Vmax = 407 V - maximum measurable voltage range of the power stage
  - Vmeasured = 303.5 – DC-Bus voltage measured with ADC

$$(Frac16)voltage\_variable = \frac{V_{MEASURED}}{V_{MAX}} = \frac{303.5}{407} = 0.7457$$

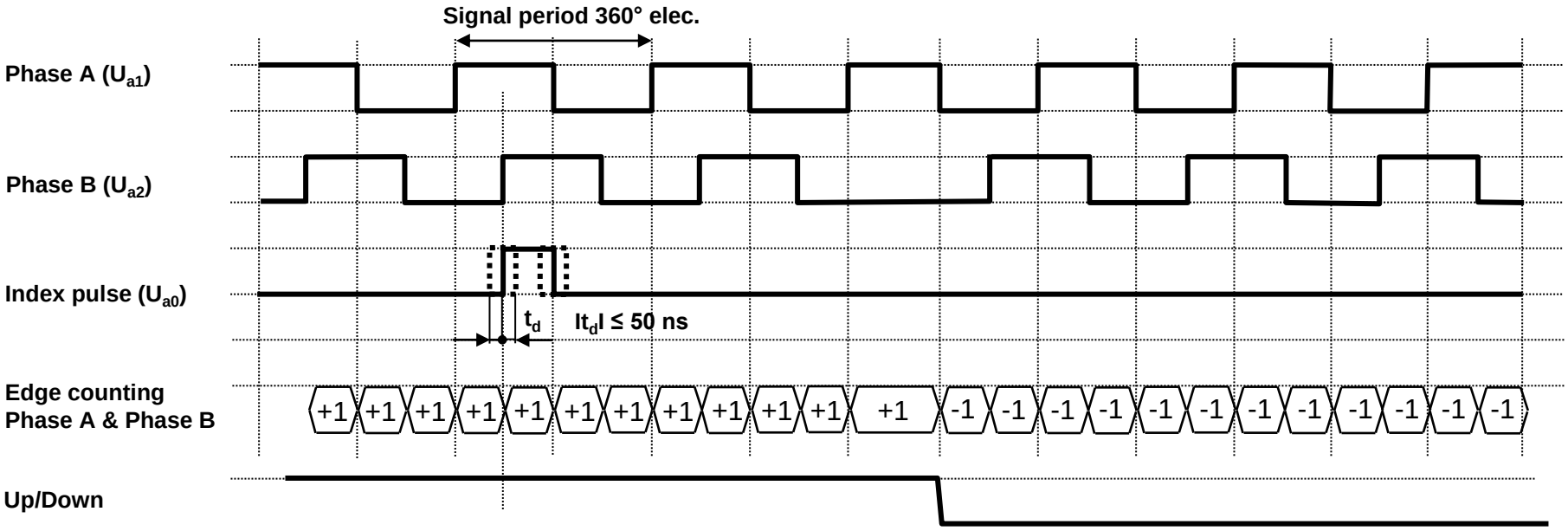
- Fractional variables are internally stored as signed 16-bit integer values

$$(Int16)voltage\_variable = (Frac16)voltage\_variable \bullet 2^{15} = 0.7457 \bullet 2^{15} = 24435$$



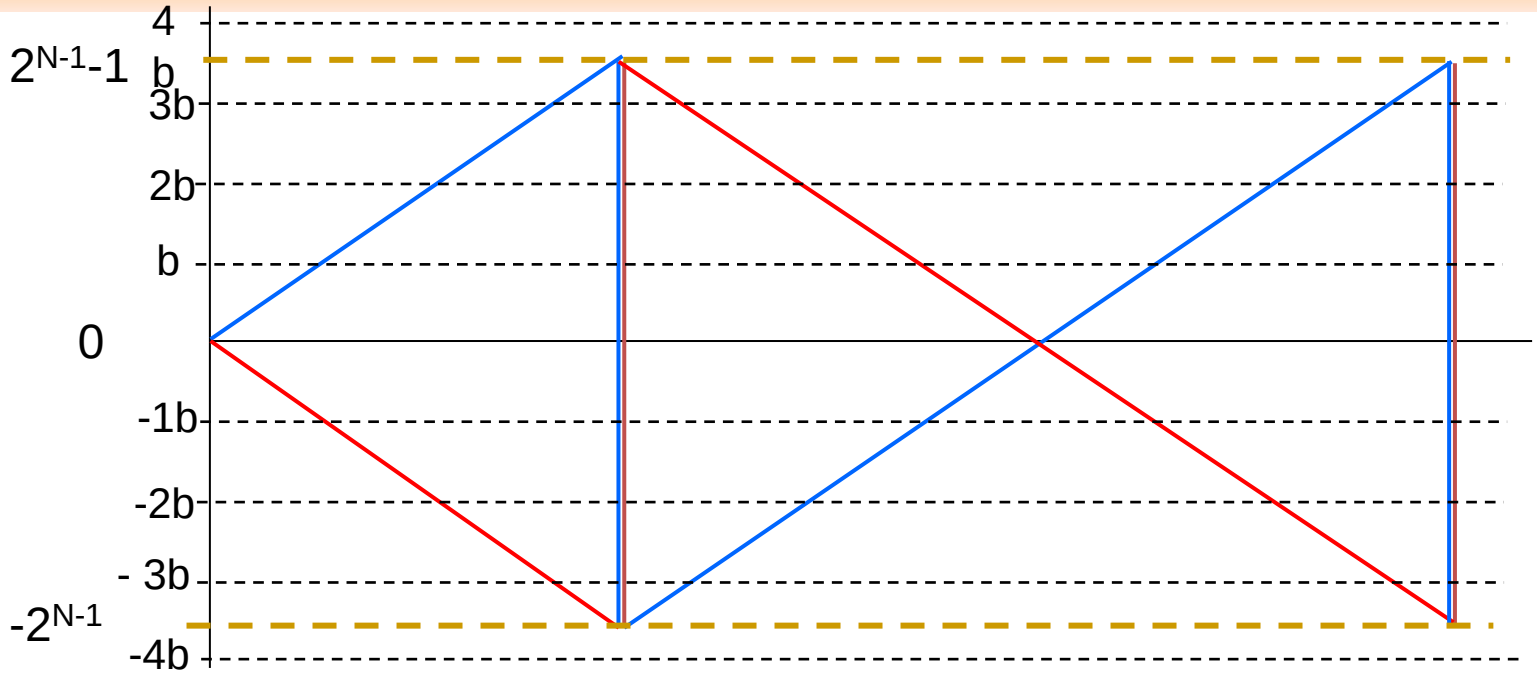
Encoder Signals Processing

Decoding the Phase A & Phase B signals generated by the incremental sensor (encoder)



- a – Number of increments per revolution, e. g. 1024 inc/rev, 512, 500, 360
- b – Number of edges per revolution, b = 4a, e.g. 1024x4 = 4096, 512x4 = 2048, 500x4 = 2000, 360x4 = 1440
- Decoding logic counts all edges of encoder signals. The quantity b is counted.

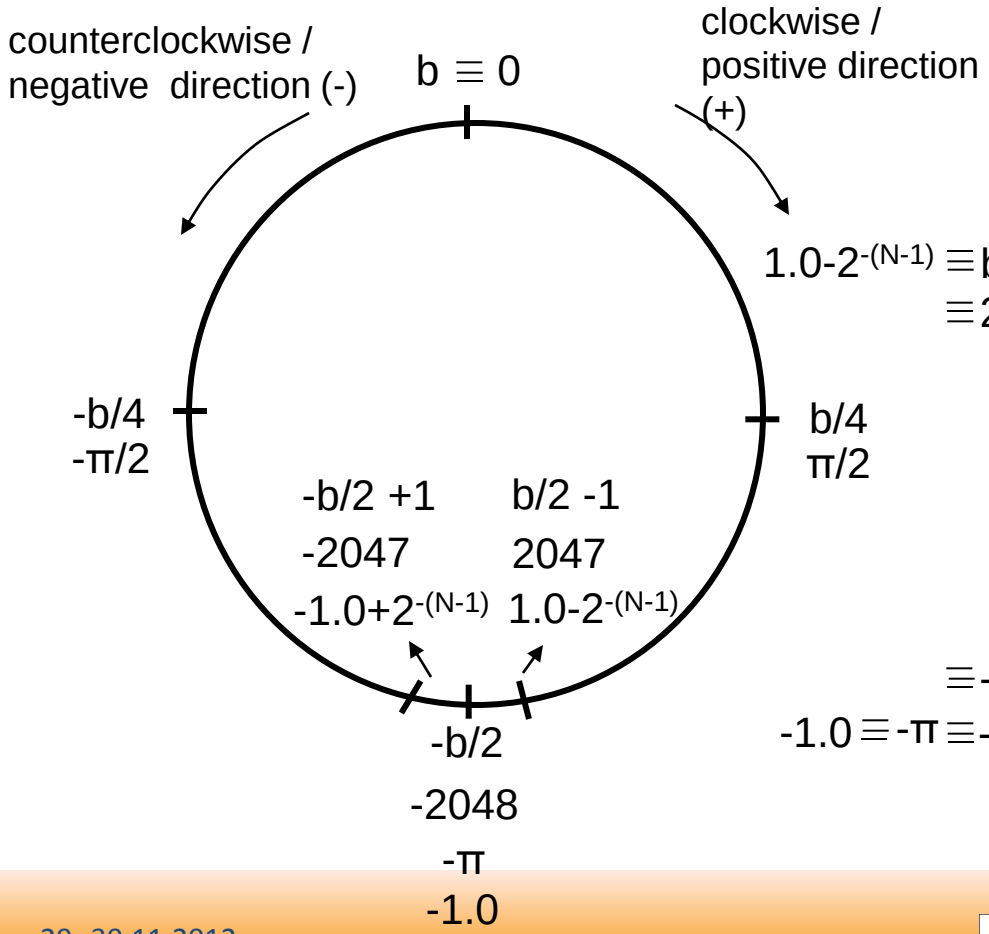
Encoder Signals Processing – Position Counter



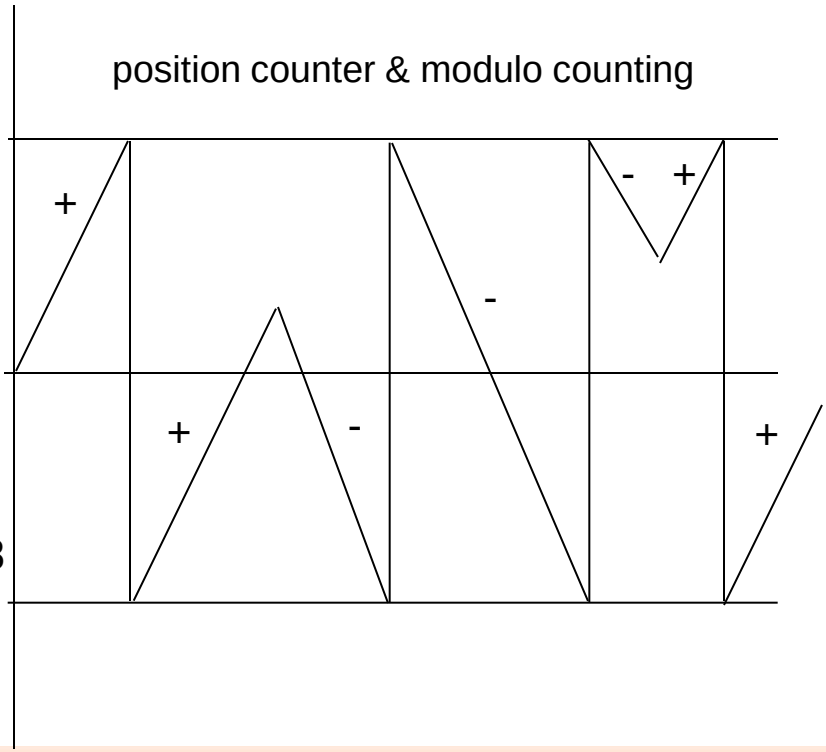
- The position counter counts signed value
- N – position counter resolution (16-bit, 32-bit)
- The blue line depicts counter when counting up to maximum positive value and then naturally overflows
- The red line depicts counter when counting down to minimal negative value and then naturally overflows
- b – number of edges corresponding to one encoder revolution

Encoder Signals Processing – Position Scaling

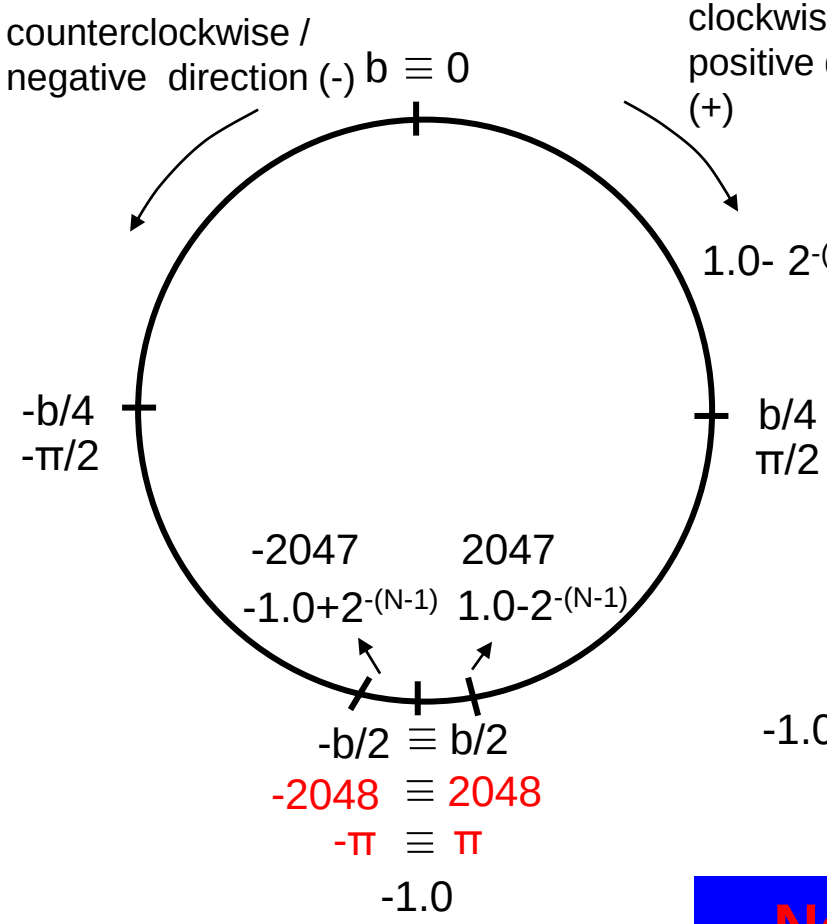
- We would like to have the result of the position value in one revolution at anytime
- The value gathered from the position counter is usually scaled to full 16-bit range which is in case of fractional arithmetic -1.0 <= position <= +1.0 – $2^{-(N-1)}$, signed integer [-2^{N-1} <= position <= $2^{(N-1)}-1$].



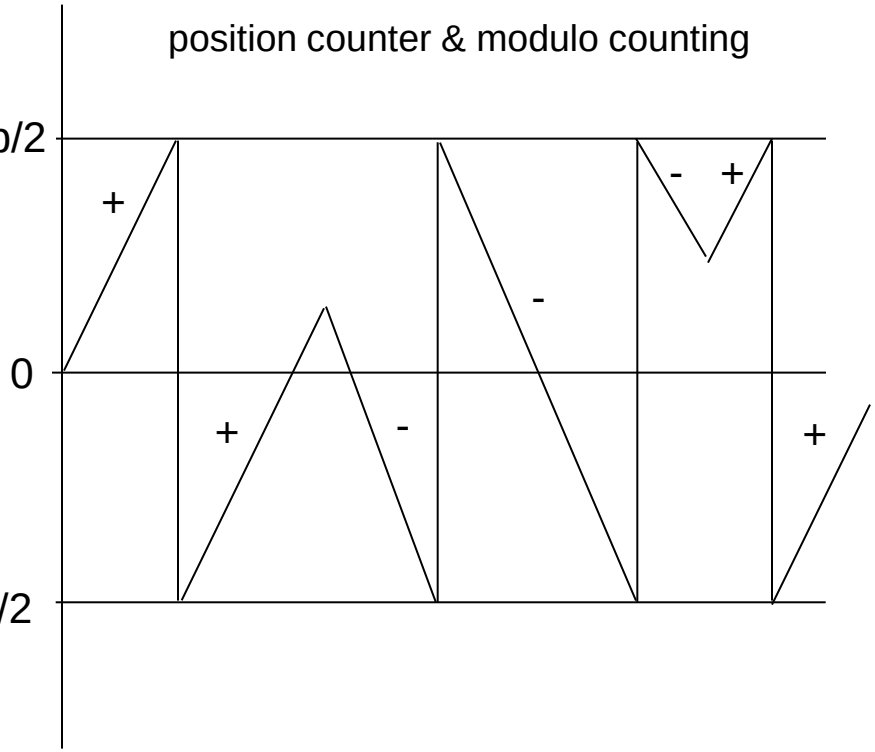
E.g. b = 1024 x 4 = 4096



Encoder Signals Processing – Position Scaling (Incorrect Approach)



- E.g. b = 1024 x 4 = 4096



Not exactly correct

Thank you