Control Law Accelerator (CLA)

Technical Training

Objectives

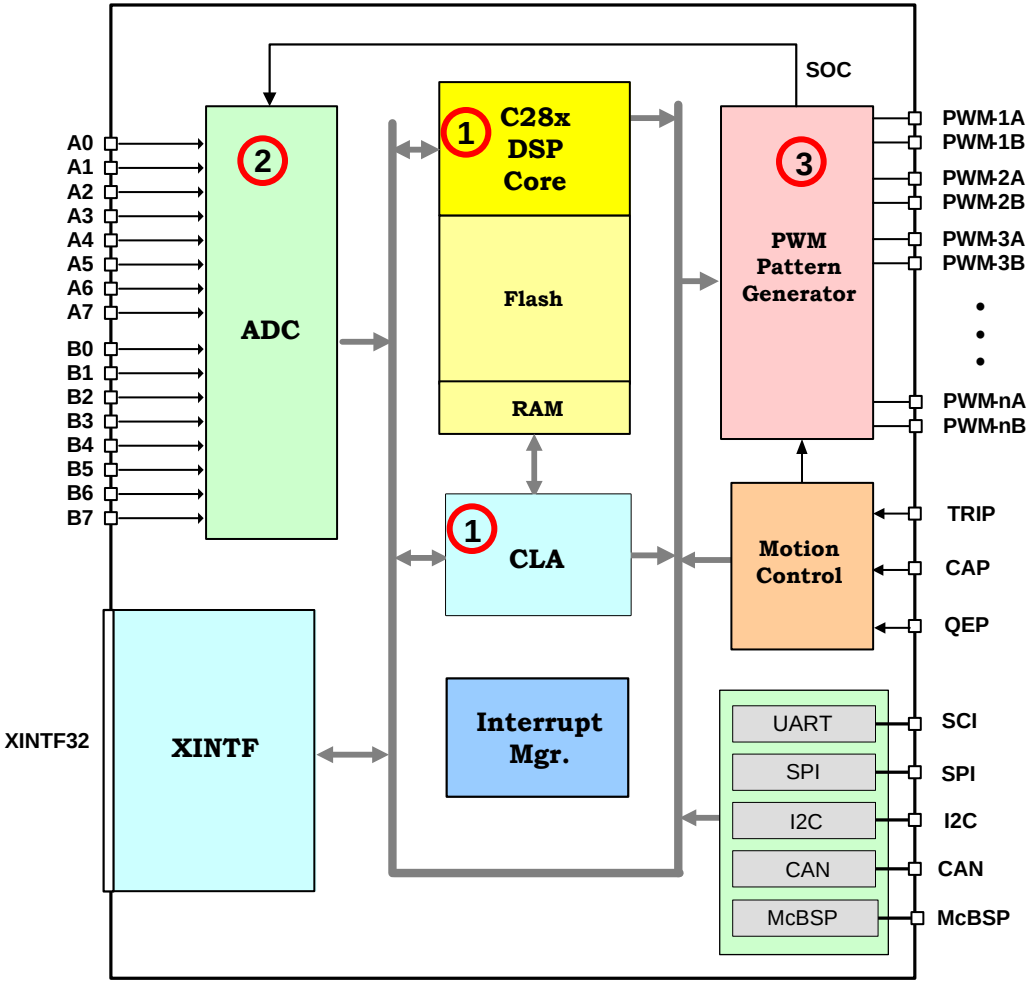
- Explain the purpose and operation of the Control Law Accelerator (CLA)
- Describe the CLA initialization procedure
- Review the CLA registers, instruction set, and programming flow
- Gain hands-on experience programming the CLA

C28x Platform Outline

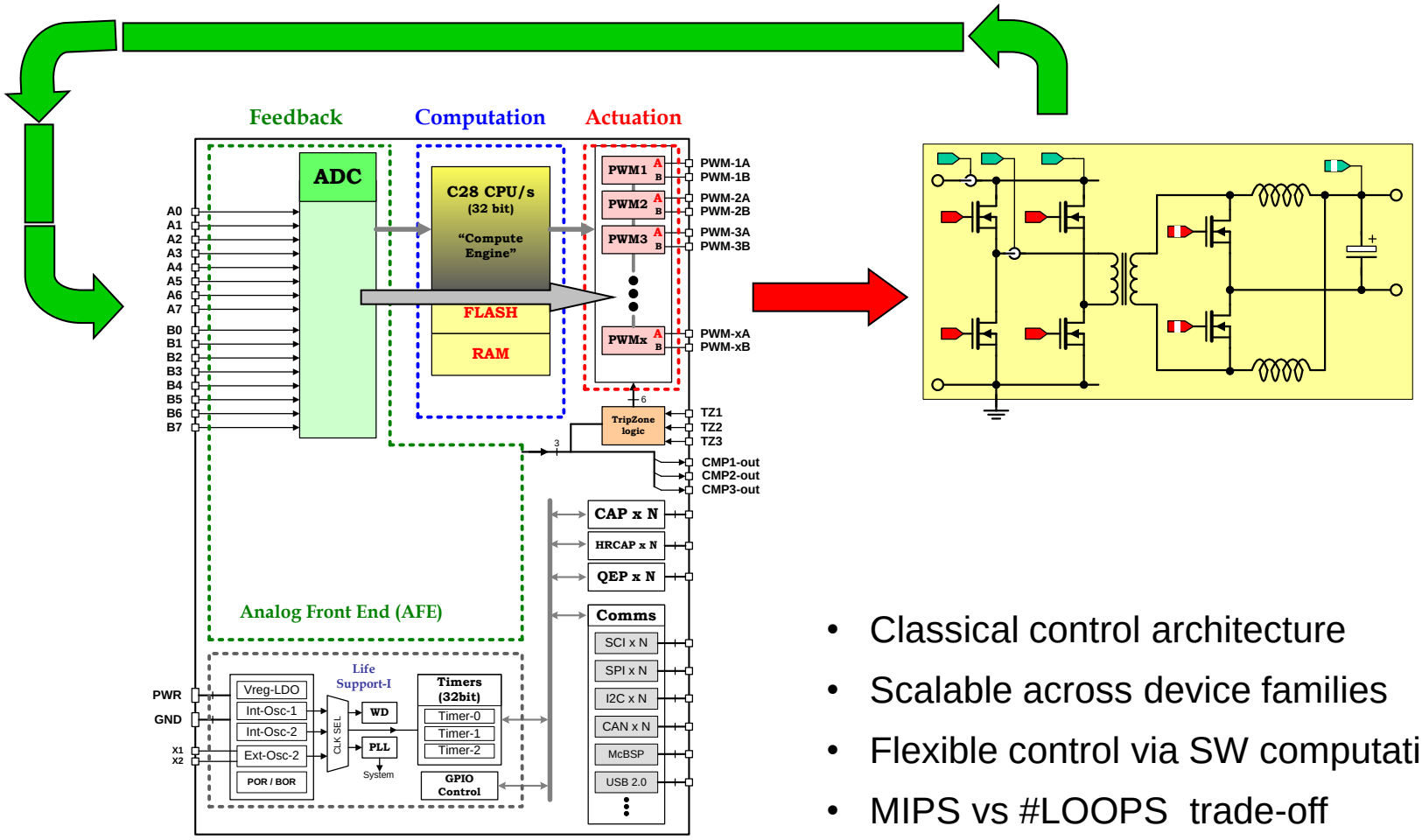
A DSP optimised for real-time control applications

Key Features

1. **High performance DSP core**
   - Low latency
   - Real-time debug mode
   - 32-bit floating-point
   - Control Law Accelerator
2. **High-speed ADC**
   - 12-bit, 12.5 MSPS
   - 16-channels, 2 x S/H
3. **Flexible PWM outputs**
   - Hardware trip inputs
   - High-resolution

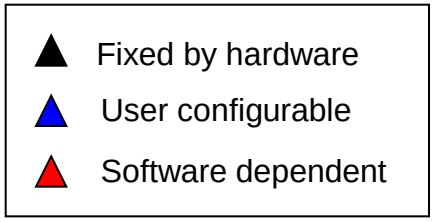


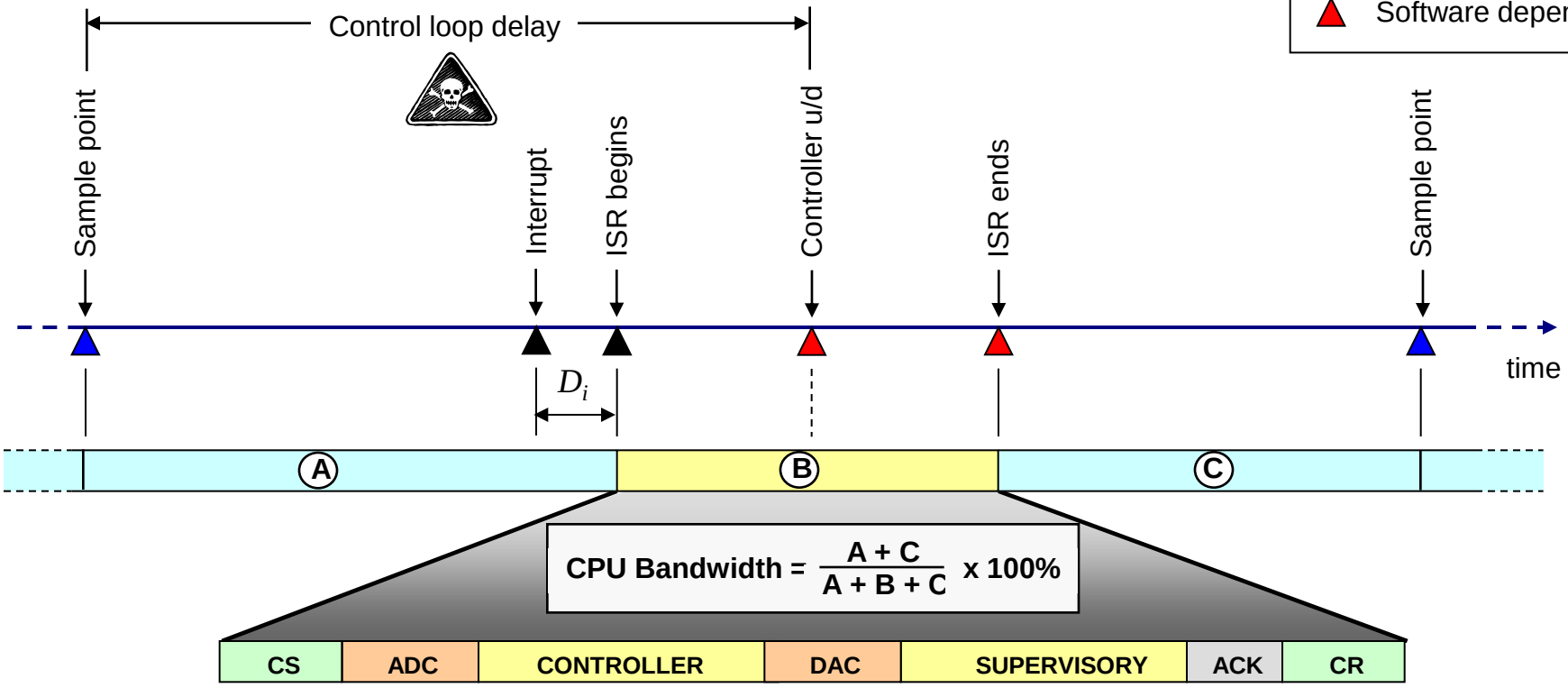
Closed Loop Control



- Classical control architecture
- Scalable across device families
- Flexible control via SW computation
- MIPS vs #LOOPS trade-off
- HW assistance via feature rich peripherals

Closed Loop Delay





Example: 9 + 3 + 27 + 5 + 23 + 1 + 9 = 77 cycles?

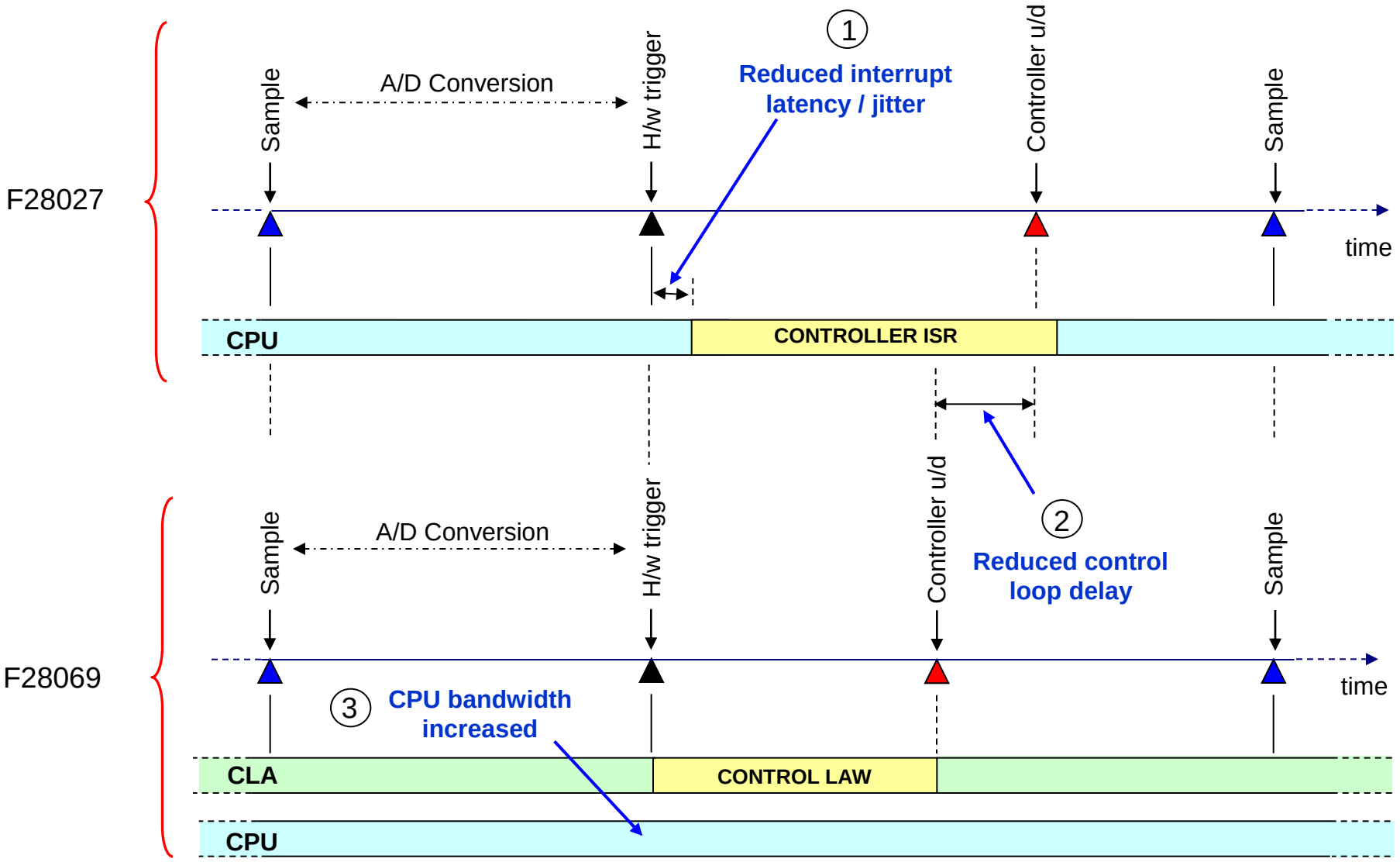
- 9 — CONTEXT SAVE
- 3 — READ ADC
- 27 — COMPUTE NEW CONTROL
- 5 — UPDATE CONTROLLER
- 23 — COMMS, WATCHDOG, ETC
- 1 — ACKNOWLEDGE INTERRUPT
- 9 — CONTEXT RESTORE

Need to determine the cycle count of this!

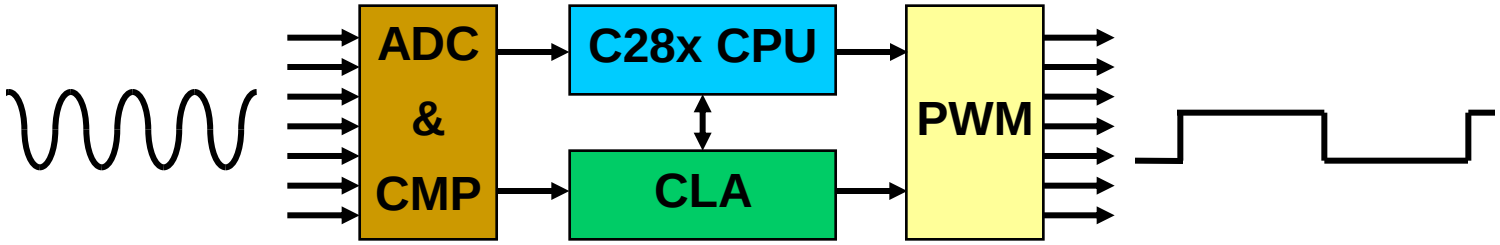
ISR code

Other code

Benefits of CLA



Control Law Accelerator (CLA)



- An independent 32-bit floating-point math accelerator
- Executes algorithms independently and in parallel with the main CPU
- Direct access to ePWM / HRPWM, eCAP, eQEP, ADC result and comparator registers
- Responds to peripheral interrupts independently of the CPU
- Frees up the CPU for other tasks (communications and diagnostics)

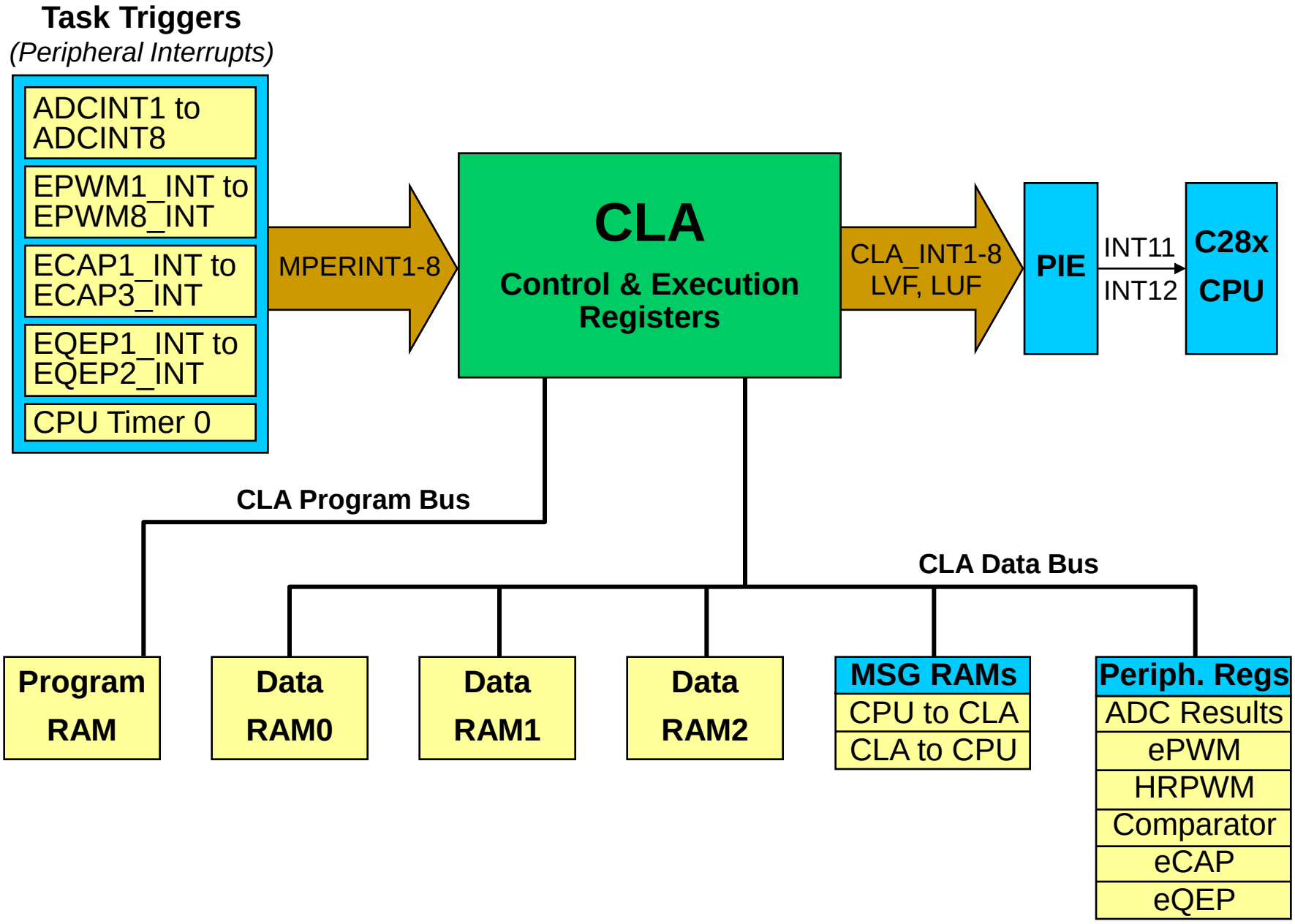
CLA Availability

- The Control Law Accelerator (CLA) is currently available on the following devices:
  - F2803x
  - F2806x
  - F2805x
- The CLA will be a standard feature of new C2000 devices

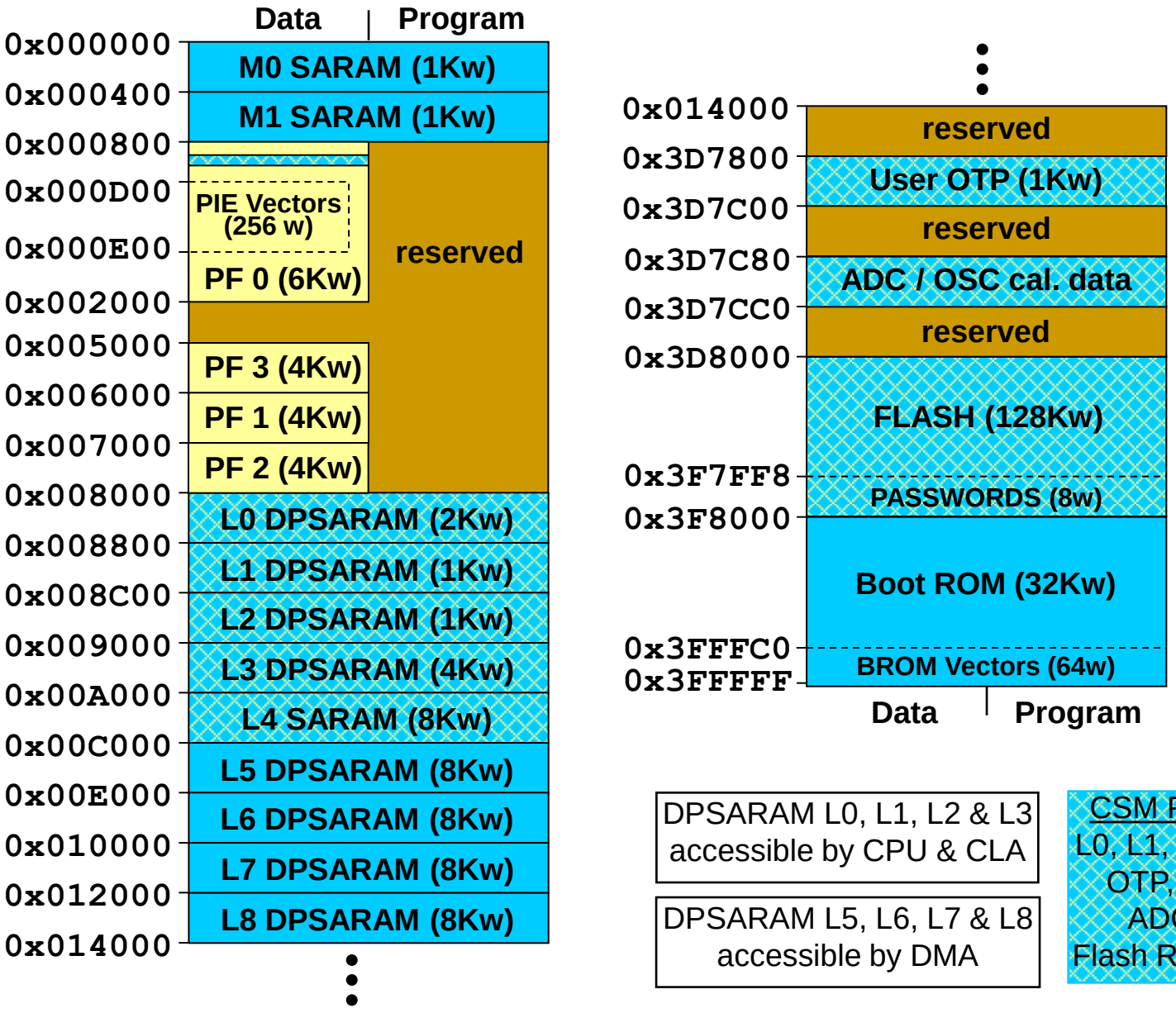
CLA vs. C28x CPU

Control Law Accelerator	C28x + Floating-Point Unit
Independent 8 Stage Pipeline	F1-D2 Shared with the C28x Pipeline
Single Cycle Math and Conversions	Math and Conversions are 2 Cycle
No Data Page Pointer. Only uses Direct & Indirect with Post-Increment	Uses C28x Addressing Modes
4 Result Registers 2 Independent Auxiliary Registers No Stack Pointer or Nested Interrupts	8 Result Registers Shares C28x Auxiliary Registers Supports Stack, Nested Interrupts
Native Delayed Branch, Call & Return Use Delay Slots to Do Extra Work No repeatable instructions	Uses C28x Branch, Call and Return Copy flags from FPU STF to C28x ST0 Repeat MACF32 & Repeat Block
Self-Contained Instruction Set Data is Passed Via Message RAMs	Instructions Superset on Top of C28x Pass Data Between FPU and C28x Regs
Supports Native Integer Operations: AND, OR, XOR, ADD/SUB, Shift	C28x Integer Operations
Programmed in C subset or Assembly	Programmed in C/C++ or Assembly
Single step moves the pipe one cycle	Single step flushes the pipeline

CLA Block Diagram



TMS320F28069 Memory Map



DPSARAM L0, L1, L2 & L3 accessible by CPU & CLA

DPSARAM L5, L6, L7 & L8 accessible by DMA

CSM Protected:
L0, L1, L2, L3, L4, OTP, FLASH, ADC CAL, Flash Regs in PF0

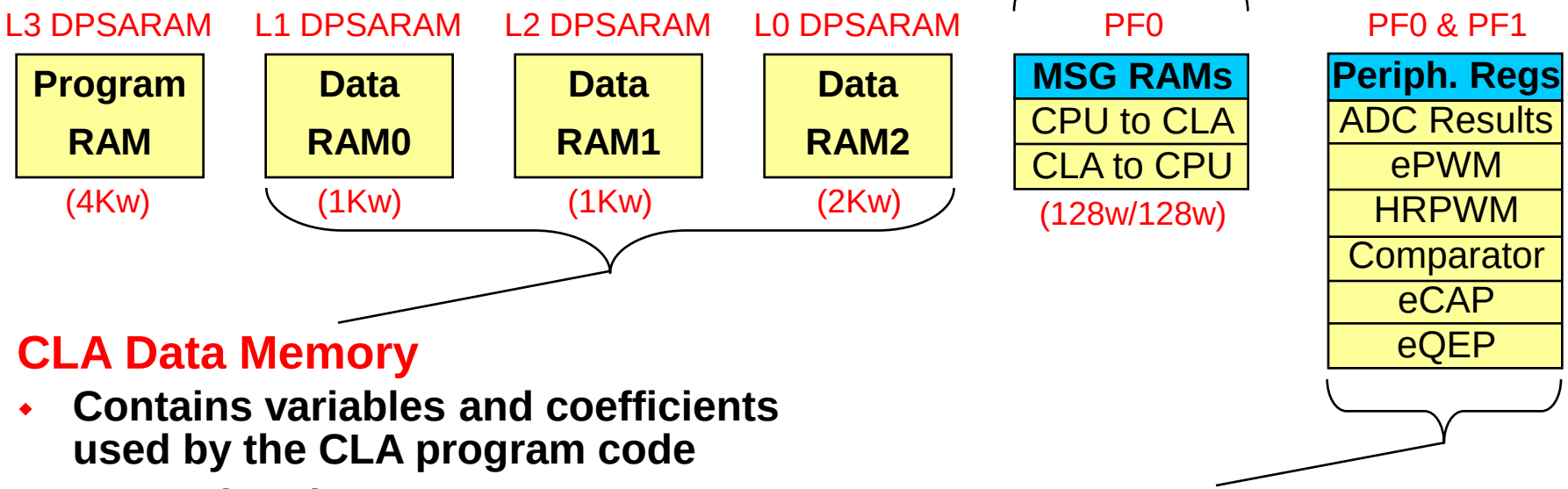
CLA Memory and Register Access

CLA Program Memory

- Contains CLA program code
- Mapped to the CPU at reset
- Initialized by the CPU

Message RAMs

- Used to pass data between the CPU and CLA
- Always mapped to both the CPU and CLA



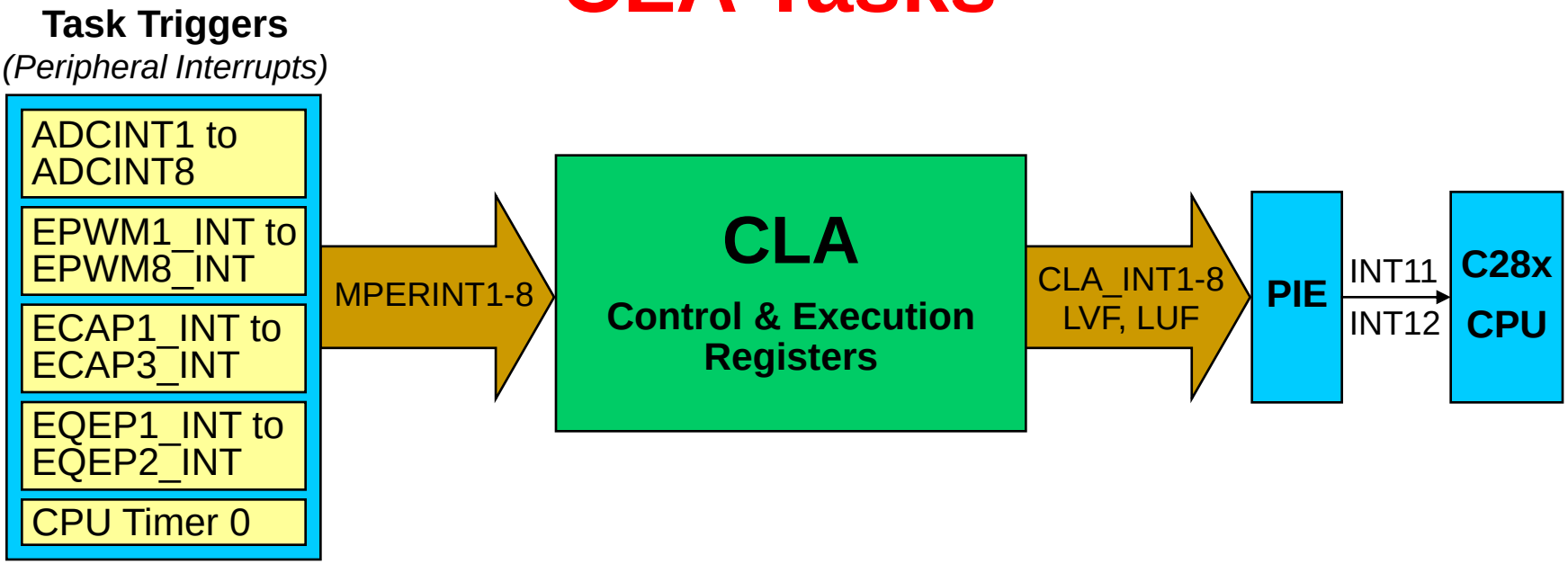
CLA Data Memory

- Contains variables and coefficients used by the CLA program code
- Mapped to the CPU at reset
- Initialized by CPU

Peripheral Reg Access

- ADC Results Regs
- ePWM (all regs)
- HRPWM (all regs)
- Comparator (all regs)
- eCAP (all regs)
- eQEP (all regs)

CLA Tasks

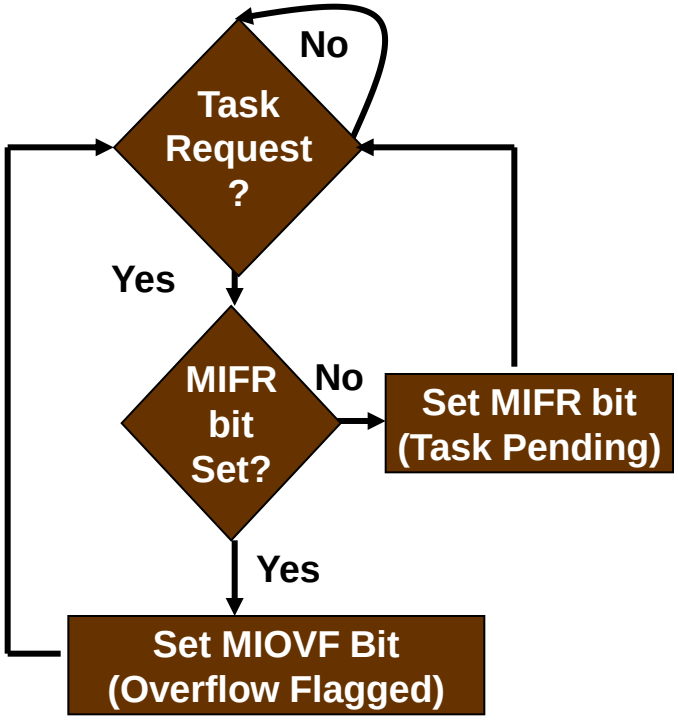


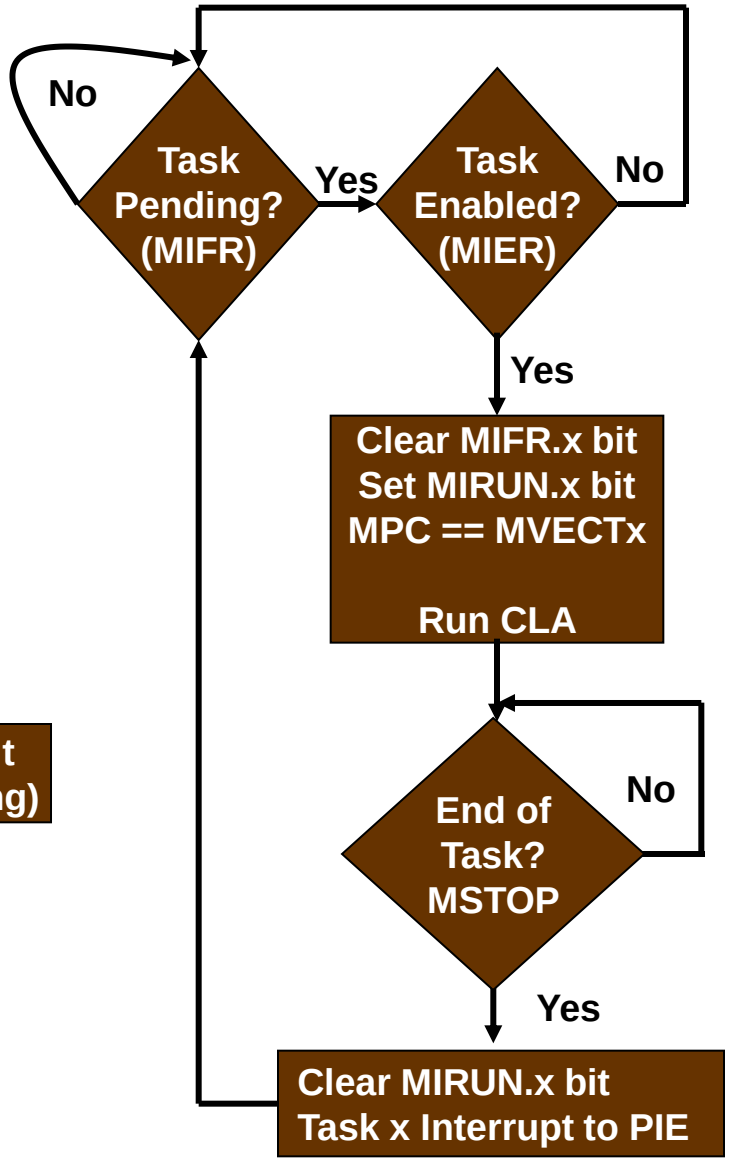
- **A *Task* is similar to an interrupt service routine**
- **CLA supports 8 Tasks (Task1-8)**
- **A task is started by a *peripheral interrupt trigger***
  - **Triggers are enabled in the MPISRCSEL1 register**
- **When a trigger occurs the CLA begins execution at the associated task vector entry (MVECT1-8)**
- **Once a task begins it runs to completion (no nesting)**

Task Flow Diagram

Task request is via software or interrupt assigned in MPISRCSEL1:

Task1: ADCINT1 or EPWM1_INT
Task2: ADCINT2 or EPWM2_INT
…
Task7: ADCINT7, EPWM7_INT, EQEP1/2_INT or ECAP1/2/3_INT
Task8: ADCINT8, CPU Timer 0, EQEP1/2_INT or ECAP1/2/3_INT





x = Highest priority task both enabled and pending

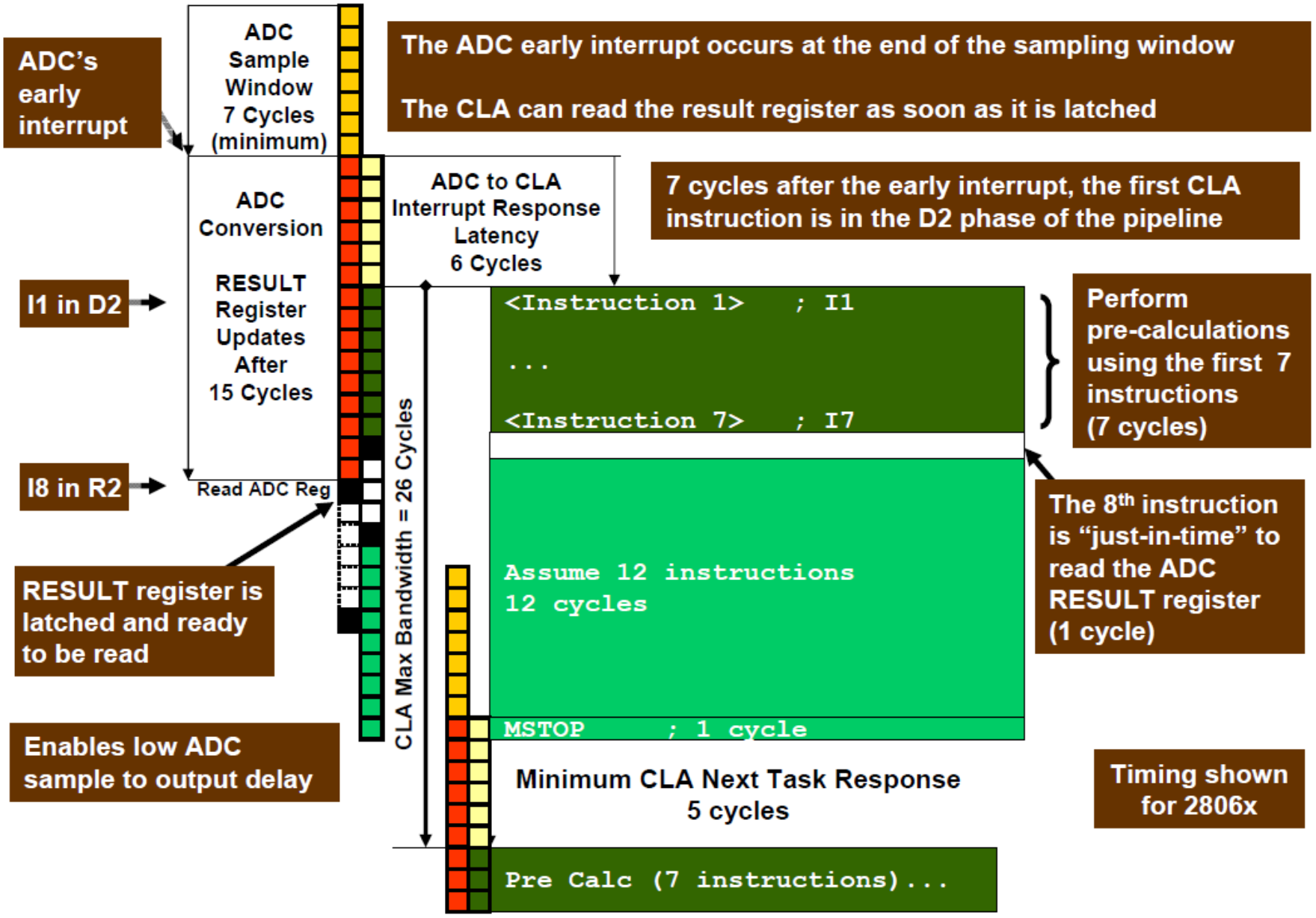
	Priority
Task1:	Highest
…	
Task8:	Lowest

The task runs to completion (No task nesting)

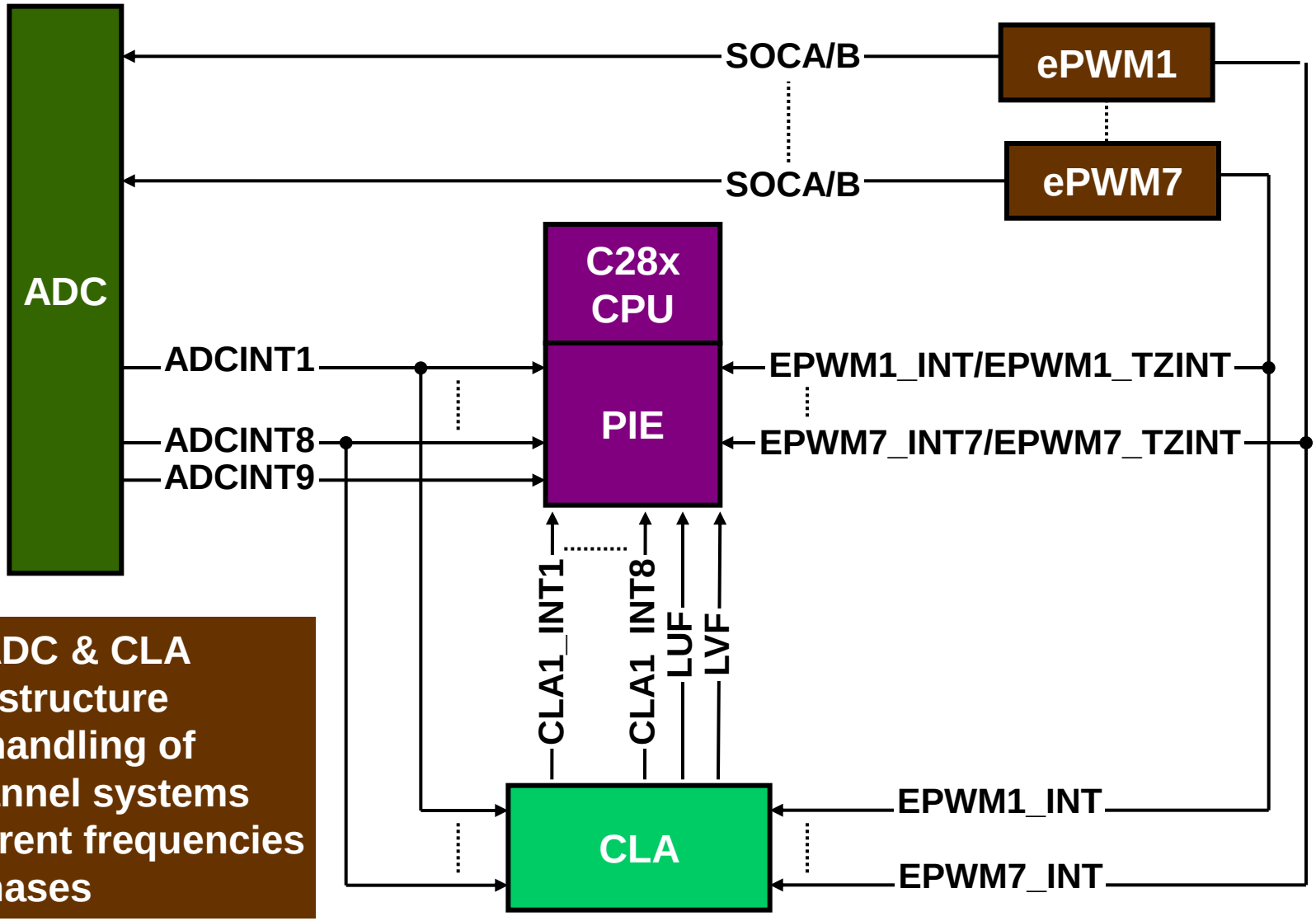
The main CPU continues code execution in parallel with the CLA

When a task completes a task-specific interrupt is sent to the PIE

ADC Sampling



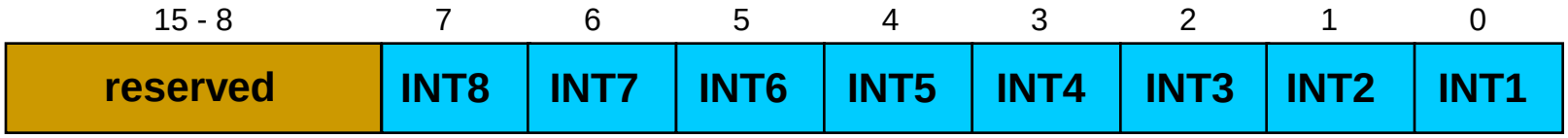
CLA Interrupts



Piccolo ADC & CLA interrupt structure enables handling of multi-channel systems with different frequencies and/or phases

Software Triggering a Task

- **Tasks can also be started by a *software trigger* using the CPU**
  - **Method #1: Write to Interrupt Force Register (MIFRC) register**




```
asm(" EALLOW");                 //enable protected register access
Cla1Regs.MIFRC.bit.INT4 = 1;    //start task 4
asm(" EDIS");                   //disable protected register access
```

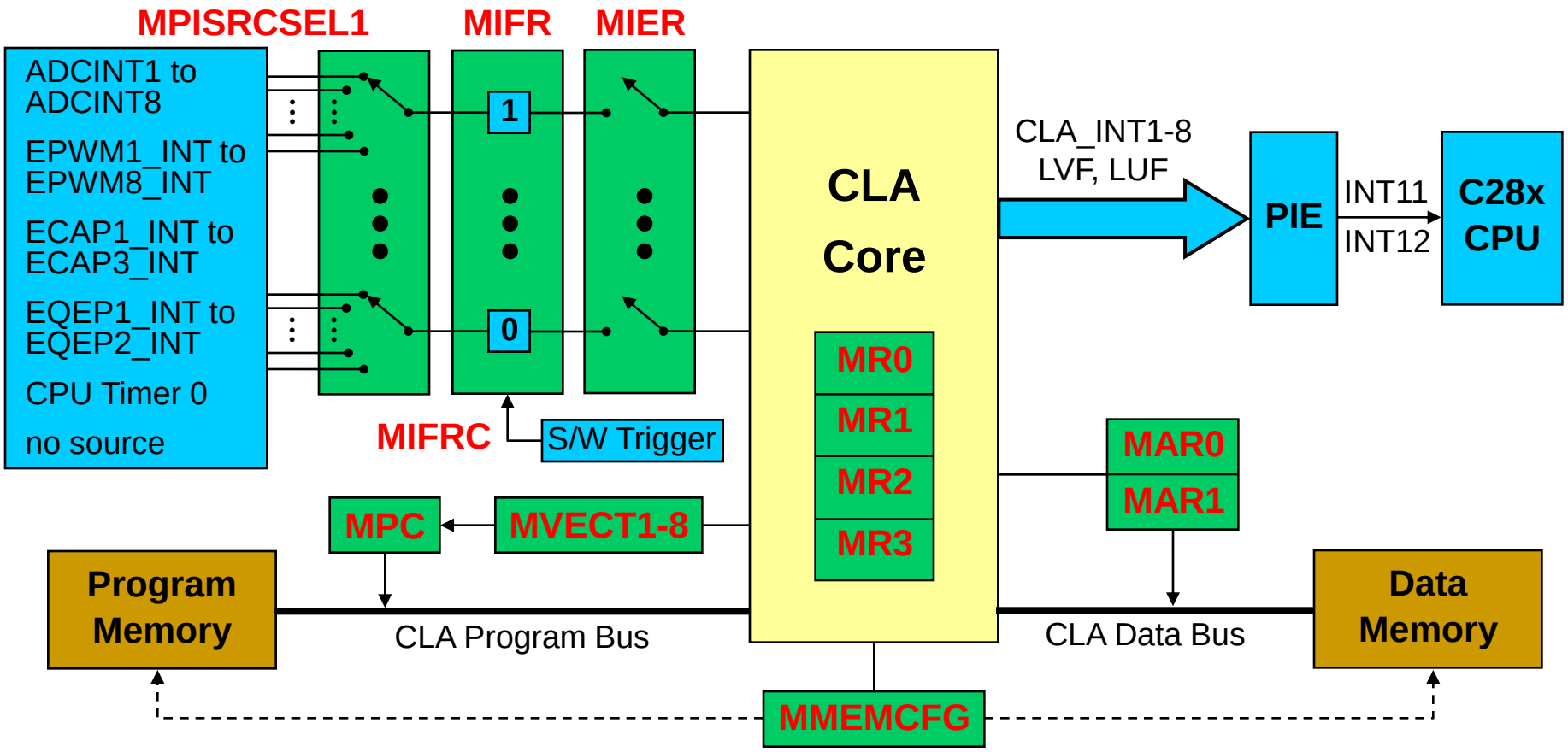
  - **Method #2: Use IACK instruction**

```
asm(" IACK #0x0008");           //set bit 3 in MIFRC to start task 4
```

More efficient – does not require EALLOW

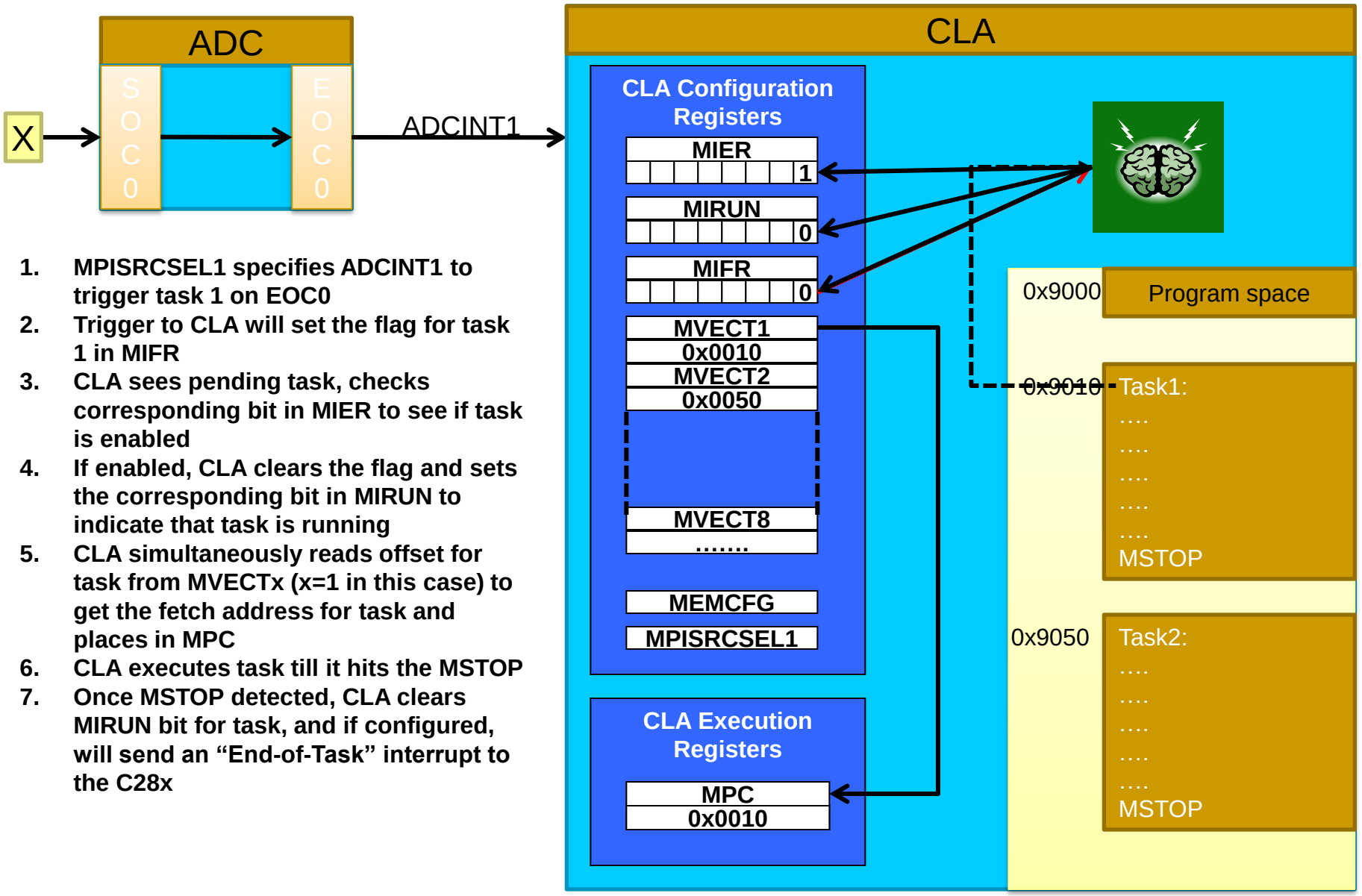
Note: Use of IACK requires Cla1Regs.MCTL.bit.IACKE = 1

CLA Control and Execution Registers



- MPISRCSEL1 – Peripheral Interrupt Source Select (Task 1-8)
- MVECT1-8 – Task Interrupt Vector (MVECT1/2/3/4/5/6/7/8)
- MMEMCFG – Memory Map Configuration (RAM2E, RAM1E, RAM0E, PROGE)
- MPC – 12-bit Program Counter (initialized by appropriate MVECTx register)
- MR0-3 – CLA Floating-Point Result Registers (32-bit)
- MAR0-1 – CLA Auxiliary Registers (16-bit)

Execution Flow Example



1. MPISRCSEL1 specifies ADCINT1 to trigger task 1 on EOC0
2. Trigger to CLA will set the flag for task 1 in MIFR
3. CLA sees pending task, checks corresponding bit in MIER to see if task is enabled
4. If enabled, CLA clears the flag and sets the corresponding bit in MIRUN to indicate that task is running
5. CLA simultaneously reads offset for task from MVECTx (x=1 in this case) to get the fetch address for task and places in MPC
6. CLA executes task till it hits the MSTOP
7. Once MSTOP detected, CLA clears MIRUN bit for task, and if configured, will send an “End-of-Task” interrupt to the C28x

CLA Registers

Cla1Regs.*register* (lab file: Cla.c)

Register	Description
MCTL	Control Register
MMEMCFG	Memory Configuration Register
MPISRCSEL1	Peripheral Interrupt Source Select 1 Register
MIFR	Interrupt Flag Register
MIER	Interrupt Enable Register
MIFRC	Interrupt Force Register
MICLR	Interrupt Flag Clear Register
MIOVF	Interrupt Overflow Flag Register
MICLROVF	Interrupt Overflow Flag Clear Register
MIRUN	Interrupt Run Status Register
MVECTx	Task x Interrupt Vector (x = 1-8)
MPC	CLA 12-bit Program Counter
MARx	CLA Auxiliary Register x (x = 0-1)
MRx	CLA Floating-Point 32-bit Result Register (x = 0-3)
MSTF	CLA Floating-Point Status Register

CLA Control Register

Cla1Regs.MCTL

IACK Enable

0 = CPU IACK instruction ignored
1 = CPU IACK instruction triggers a task

Hard Reset

0 = no effect
1 = CLA reset (registers set to default state)

Soft Reset

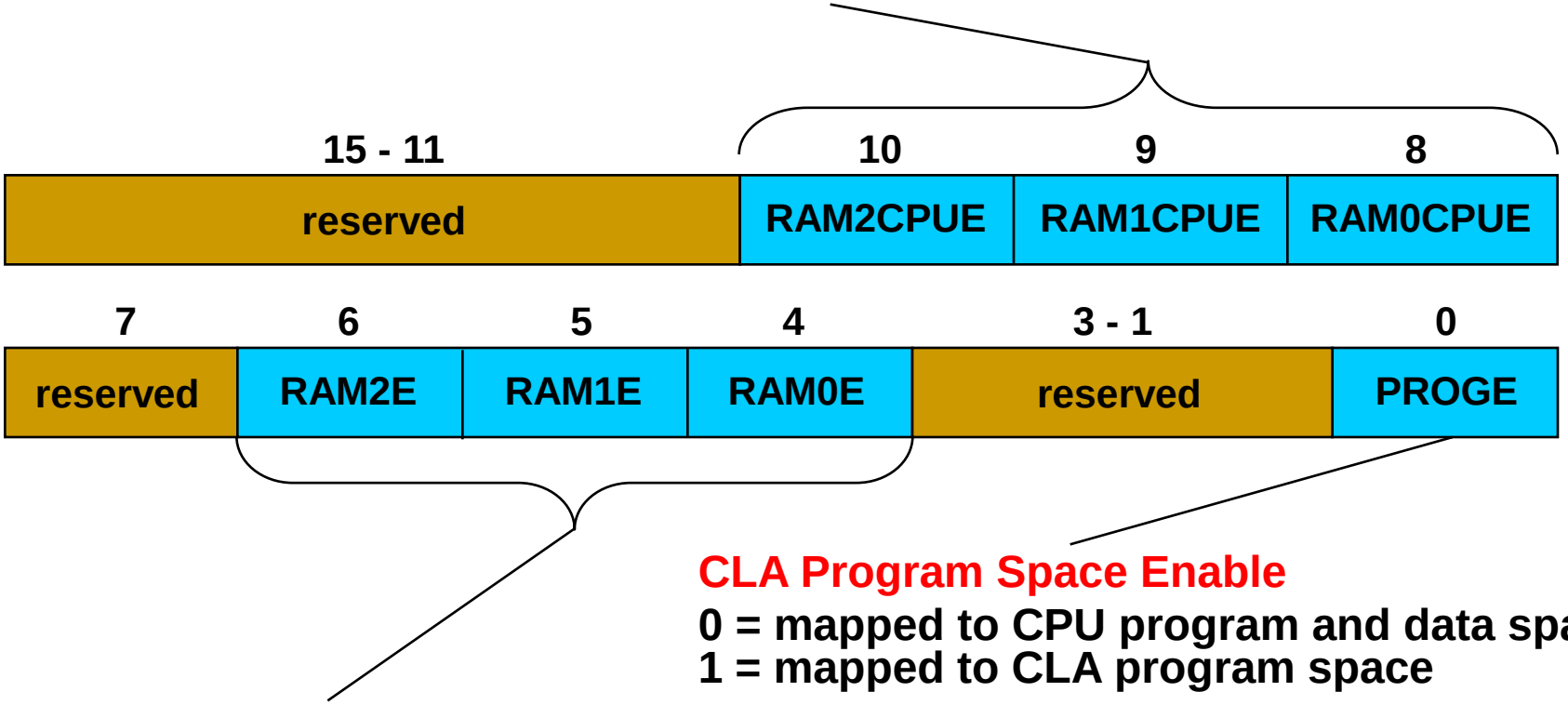
0 = no effect
1 = CLA reset (stop current task)

CLA Memory Configuration Register

Cla1Regs.MMEMCFG

CLA Data RAM2 / RAM1 / RAM0 CPU Access Enable

0 = mapped as RAM2E / RAM1E / RAM0
1 = CPU access to RAM while mapped to CLA data space



CLA Program Space Enable

0 = mapped to CPU program and data space
1 = mapped to CLA program space

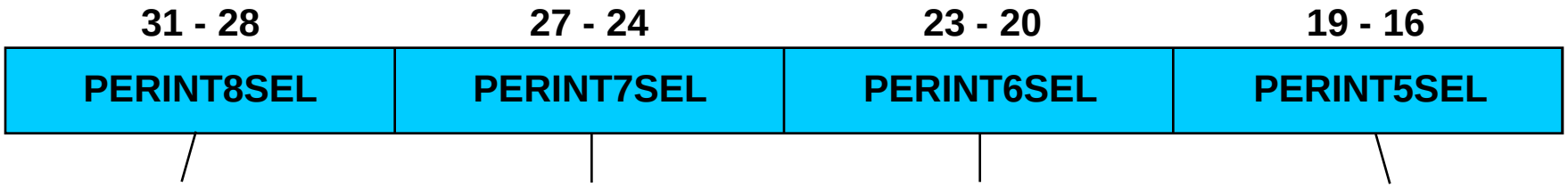
CLA Data RAM2 / RAM1 / RAM0 Enable

0 = mapped to CPU program and data space
1 = mapped to CLA data space

CLA Peripheral Interrupt Source Select 1 Register

Cla1Regs.MPISRCSEL1

Upper Register:



Task 8 Peripheral Interrupt Input	Task 7 Peripheral Interrupt Input	Task 6 Peripheral Interrupt Input	Task 5 Peripheral Interrupt Input
0000 = ADCINT8	0000 = ADCINT7	0000 = ADCINT6	0000 = ADCINT5
0010 = CPU Timer 0	0010 = ePWM7	0010 = ePWM6	0010 = ePWM5
0100 = eQEP1	0100 = eQEP1	0100 = eQEP1	0100 = eQEP1
0101 = eQEP2	0101 = eQEP2	0101 = eQEP2	0101 = eQEP2
1000 = eCAP1	1000 = eCAP1	1000 = eCAP1	1000 = eCAP1
1001 = eCAP2	1001 = eCAP2	1001 = eCAP2	1001 = eCAP2
1010 = eCAP3	1010 = eCAP3	1010 = eCAP3	1010 = eCAP3
other = no source	other = no source	other = no source	other = no source

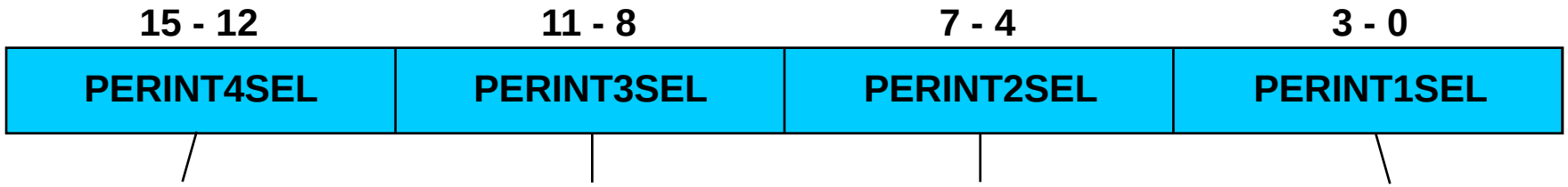
Note: select ‘no source’ if task is generated by software

0000 = Default

CLA Peripheral Interrupt Source Select 1 Register

Cla1Regs.MPISRCSEL1

Lower Register:



Task 4 Peripheral Interrupt Input

0000 = ADCINT4
0010 = ePWM4
0100 = eQEP1
0101 = eQEP2
1000 = eCAP1
1001 = eCAP2
1010 = eCAP3
other = no source

Task 3 Peripheral Interrupt Input

0000 = ADCINT3
0010 = ePWM3
xxx1 = no source

Task 2 Peripheral Interrupt Input

0000 = ADCINT2
0010 = ePWM2
xxx1 = no source

Task 1 Peripheral Interrupt Input

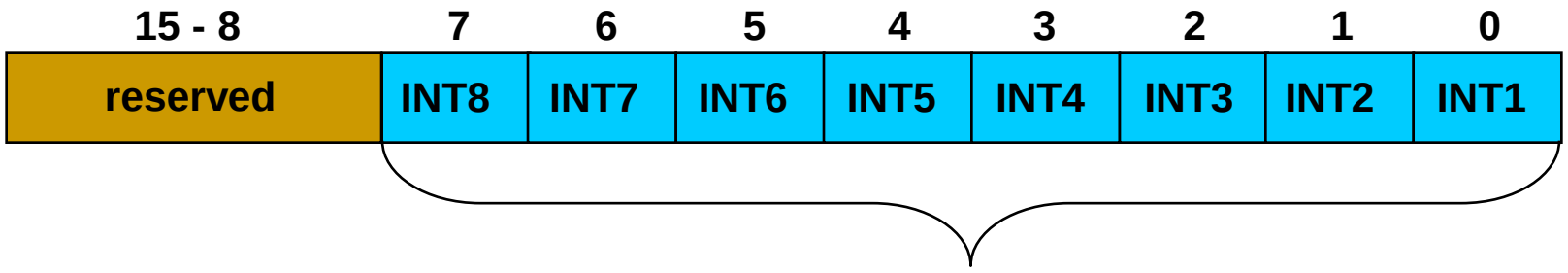
0000 = ADCINT1
0010 = ePWM1
xxx1 = no source

Note: select ‘no source’ if task is generated by software

0000 = Default

CLA Interrupt Enable Register

Cla1Regs.MIER



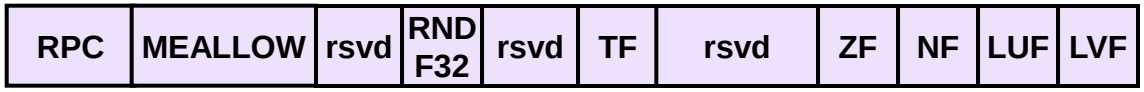
0 = task interrupt disable (default)

1 = task interrupt enable

```
#include "F2806x_Device.h"
  Cla1Regs.MIER.bit.INT2 = 1;  //enable Task 2 interrupt
  Cla1Regs.MIER.all = 0x0028;  //enable Task 6 and 4 interrupts
```

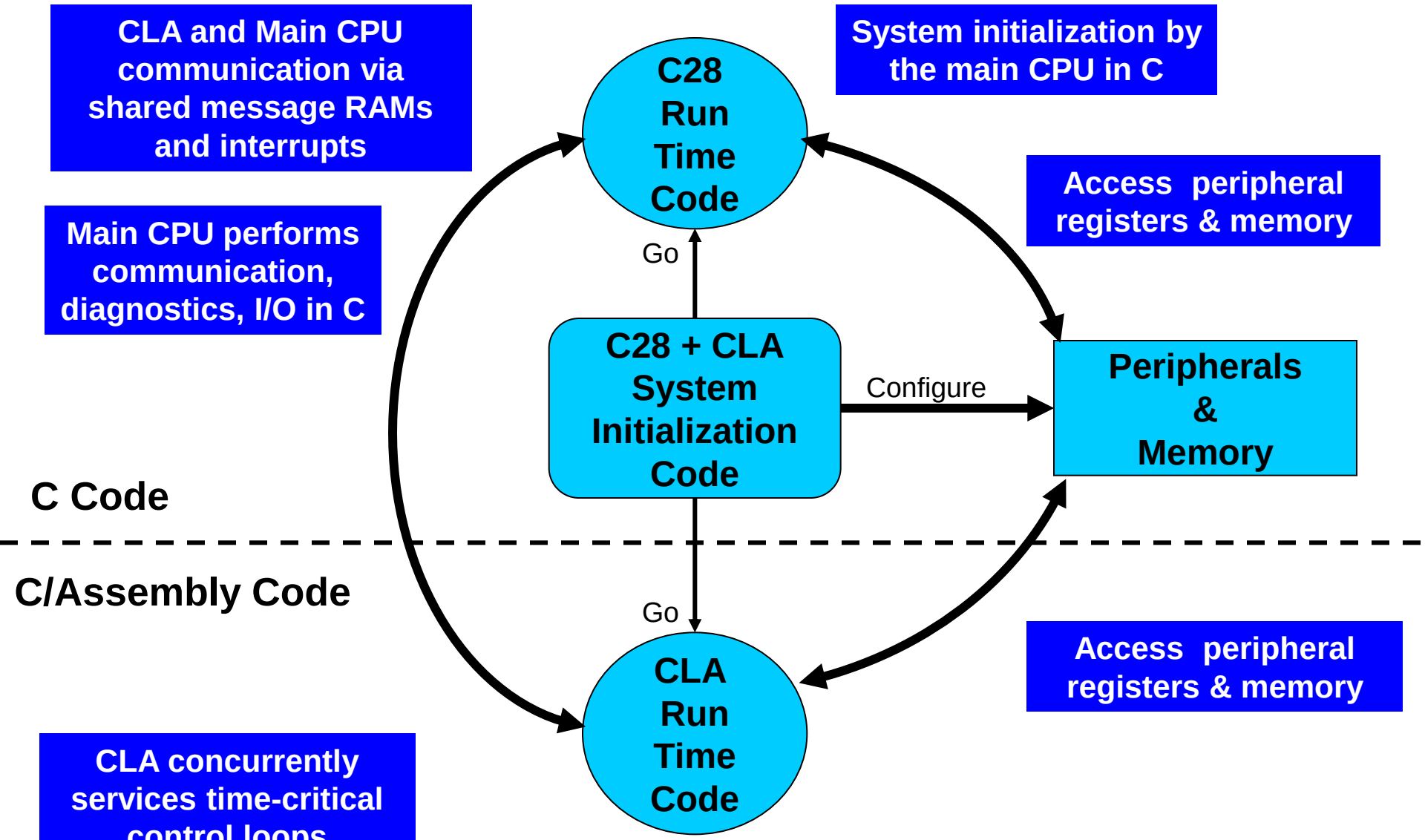
CLA Status Flags

CLA Status Register MSTF (32-bits)



LVF LUF	Latched Overflow and Underflow	Float math: MMPYF32, MADDF32, 1/x etc. Connected to the PIE for debug
ZF NF	Negative and Zero	Float move operations to registers. Result of compare, min/max, absolute, negative Integer result of integer operations (MAND32, MOR32, SUB32, MLSR32 etc.)
TF	Test Flag	MTESTTF Instruction
RNDF32	Rounding Mode	To Zero (truncate) or To Nearest (even)
MEALLOW	Write Protection	Enable/disable CLA writes to “EALLOW” protected registers
RPC	Return Program Counter	Call and return: MCNDD, MRCNDD Use store/load MSTF instructions to nest calls

CLA Code Partitioning



CLA Initialization

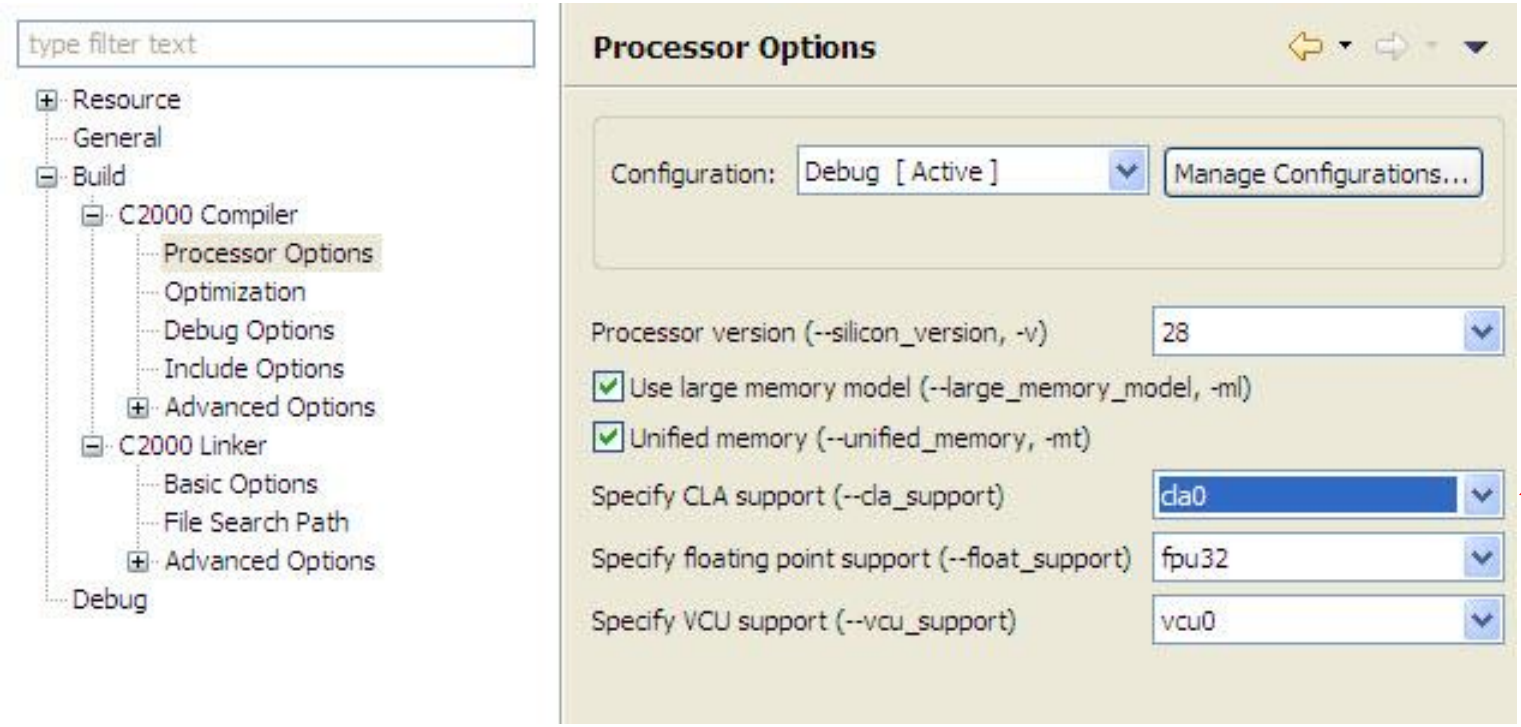
CLA initialization is performed by the CPU using C code (typically done with the Peripheral Register Header Files)

1. **Copy CLA task code from flash to CLA program RAM**
2. **Initialize CLA data RAMs, as needed**
   - Populate with data coefficients, constants, etc.
3. **Configure the CLA registers**
   - Enable the CLA clock (PCLKCR3 register)
   - Populate the CLA task interrupt vectors (MVECT1-8 registers)
   - Select the desired task interrupt sources (MPISRCSEL1 register)
   - If desired, enable IACK to start task using software (avoids EALLOW)
   - Map CLA program RAM and data RAMs to CLA space
4. **Configure desired CLA task completion interrupts in the PIE**
5. **Enable CLA tasks triggers in the MIER register**
6. **Initialize the desired peripherals to trigger the CLA tasks**

Data is passed between the CLA and CPU via message RAMs

Enabling CLA Support in CCS

- **Set the "Specify CLA support" project option to 'cla0'**
- **When creating a new CCS project, choosing a device variant that has the CLA will automatically select this option, so normally no user action is required**



CLA Task Programming

- Can be written in C or assembly code
- Assembly code will give best performance for time-critical tasks
- Writing in assembly may not be so bad!
  - CLA programs in floating point
  - Often not that much code in a task
- Commonly, the user will use assembly for critical tasks, and C for non-critical tasks

CLA C Language Implementation

- **Supports C only** (no C++ or GCC extension support)
- **Different data type sizes than C28x CPU**
  - **No support for 64-bit integer or 64-bit floating point**

TYPE	CPU	CLA
char, short	16 bit	16 bit
int	16 bit	32 bit
long	32 bit	32 bit
long long	64 bit	32 bit
float, double	32 bit	32 bit
long double	64 bit	32 bit
pointers	32 bit	16 bit

- **CLA architecture is designed for 32-bit data types**
  - **16-bit computations incur overhead for sign-extension**
  - **Primarily used for reading and writing to 16-bit peripheral registers**

CLA C Language Restrictions (1 of 2)

- **CLA C compiler does not support:**
  - **Initialized global and static data**

```
int x;      // valid
int x=5;    // not valid
```

  - **Initialized variables need to be manually handled by an initialization task**
  - **More than 2 levels of function nesting**
  - **Function with more than two arguments**
  - **Recursive function calls**
  - **Function pointers**

CLA C Language Restrictions (2 of 2)

- **CLA C compiler does not support:**
  - **Certain fundamental math operations**
    - **integer division:** `z = x/y;`
    - **modulus (remainder):** `z = x%y;`
    - **unsigned 32-bit integer compares**

```
Uint32 i;   if(i < 10) {…}   // not valid
int32 i;    if(i < 10) {…}   // valid
Uint16 i;   if(i < 10) {…}   // valid
int16 i;    if(i < 10) {…}   // valid
float32 x;  if(x < 10) {…}   // valid
```

  - **C Standard math library functions**

C Intrinsics

float __meisqrtf32(float)	unsigned short __mf32toui16r(float)
float __meinvf32(float)	__mdebugstop()
float __mminf32(float, float)	__meallow()
float __mmaxf32(float, float)	__medis()
void __mswapf(float, float)	__msetflg(unsigned short, unsigned short)
short __mf32toi16r(float)	__mnop()
float __mfracf32(float)	abs()
float __sqrt(float)	fabs()

CLA Compiler Scratchpad Memory Area

- For local and compiler temporary variables
- Static allocation, used instead of a stack
- Defined in linker command file

Lab.cmd

```
 CLA_SCRATCHPAD_SIZE = 0x100;
 --undef_sym=__cla_scratchpad_end
 --undef_sym=__cla_scratchpad_start
MEMORY
{
     :
}

SECTIONS
{
     :
 Cla1Prog      :> L3DPSARAM,                PAGE = 0
                  RUN_START(_Cla1Prog_Start)

 CLAscratch    :{*.obj(CLAscratch
                  . += CLA_SCRATCHPAD_SIZE;
                  *.obj(CLAscratch_end)
                 } > L2DPSARAM,             PAGE = 1
}
```

Linker defined symbol specifies size for scratchpad area

Scratchpad area accessed directly using symbols

All CLA C code will be placed in the section Cla1Prog

Symbol used to define the start of CLA program memory

Must allocate to memory section that CLA has write access

CLA Task C Code Example

ClaTasks_C.cla

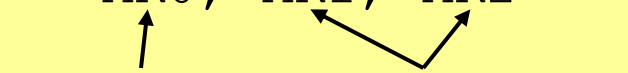
```
#include "Lab.h"
;-------------------------------------
interrupt void Cla1Task1 (void)
{
      ⋮
    __mdebugstop();
      ⋮
    xDelay[0] = (float32)AdcResult.ADCRESULT0;
    Y = coeffs[4] * xDelay[4];
    xDelay[4] = xDelay[3];
      ⋮
    xDelay[1] = xDelay[0];
    Y = Y + coeffs[0] * xDelay[0];
    ClaFilteredOutput = (Uint16)Y;
}
;-------------------------------------
interrupt void Cla1Task2 (void)
{
      ⋮
}
;-------------------------------------
```

- **.cla extension causes the c2000 compiler to invoke the CLA compiler**
- **All code within this file is placed in the section "Cla1Prog"**
- **C Peripheral Register Header File references can be used in CLA C and assembly code**
- **Closing braces are replaced with MSTOP instructions when compiled**

CLA Assembly Language Implementation

- **Same assembly instruction format as the C28x and C28x+FPU**
  - **Destination operand is always on the left**
  - **Same mnemonics as C28x+FPU but with a leading "M"**

CPU:	`MPY`	`ACC, T, loc16`
FPU:	`MPYF32`	`R0H, R1H, R2H`
CLA:	`MMPYF32`	`MR0, MR1, MR2`



CLA Assembly Instruction Overview

Type	Example		Cycles
Load (Conditional)	MMOV32	MRa,mem32{,CONDF}	1
Store	MMOV32	mem32,MRa	1
Load with Data Move	MMOVD32	MRa,mem32	1
Store/Load MSTF	MMOV32	MSTF,mem32	1
Compare, Min, Max	MCMPF32	MRa,MRb	1
Absolute, Negative Value	MABSF32	MRa,MRb	1
Unsigned Integer to Float	MUI16TOF32	MRa,mem16	1
Integer to Float	MI32TOF32	MRa,mem32	1
Float to Integer & Round	MF32TOI16R	MRa,MRb	1
Float to Integer	MF32TOI32	MRa,MRb	1
Multiply, Add, Subtract	MMPYF32	MRa,MRb,MRc	1
1/X (16-bit Accurate)	MEINVF32	MRa,MRb	1
1/Sqrt(x) (16-bit Accurate)	MEISQRTF32	MRa,MRb	1
Integer Load/Store	MMOV16	MRa,mem16	1
Load/Store Auxiliary Register	MMOV16	MAR,mem16	1
Branch/Call/Return Conditional Delayed	MBCNDD	16bitdest {,CNDF}	1-7
Integer Bitwise AND, OR, XOR	MAND32	MRa,MRb,MRc	1
Integer Add and Subtract	MSUB32	MRa,MRb,MRc	1
Integer Shifts	MLSR32	MRa,#SHIFT	1
Write Protection Enable/Disable	MEALLOW		1
Halt Code or End Task	MSTOP		1
No Operation	MNOP		1

CLA Assembly Parallel Instructions

- **Parallel bars indicate a parallel instruction**
- **Parallel instructions operate as a single instruction with a single opcode and performs two operations**
  - **Example: Add + Parallel Store**

```
     MADDF32 MR3, MR3, MR1
||   MMOV32  @_Var, MR3
```

Instruction	Example	Cycles
Multiply & Parallel Add/Subtract	`MMPYF32 MRa,MRb,MRc` `|| MSUBF32 MRd,MRe,MRf`	1
Multiply, Add, Subtract & Parallel Store	`MADDF32 MRa,MRb,MRc` `|| MMOV32 mem32,MRe`	1
Multiply, Add, Subtract, MAC & Parallel Load	`MADDF32 MRa,MRb,MRc` `|| MMOV32 MRe, mem32`	1

Both operations complete in a single cycle

Multiply + Parallel Store

```
; Before: MR0 = 2.0, MR1 = 3.0, MR2 = 10.0

   MMPYF32 MR2, MR1, MR0 ; 1/1 instruction
|| MMOV32  @_X, MR2

   <any instruction>

; After: MR2 = MR1 * MR0 = 3.0 * 2.0
;        @_X = 10.0
```

Both the math operation and store complete in 1 cycle

Parallel Instruction:
MMOV32 uses the value of MR2 before the MMPY32 update!

CLA Assembly Addressing Modes

- **Two addressing modes: *Direct* and *Indirect***
- **Both modes can access the low 64Kw of memory only:**
  - **All of the CLA data space**
  - **Both message RAMs**
  - **Shared peripheral registers**

- **Direct –** *Populates opcode field with 16-bit address of the variable*

```
example 1:      MMOV32  MR1, @_VarA
example 2:      MMOV32  MR1, @_EPwm1Regs.CMPA.all
```

- **Indirect –** *Uses the address in MAR0 or MAR1 to access memory; after the read or write MAR0/MAR1 is incremented by a 16 bit signed value*

```
example 1:      MMOV32  MR0, *MAR0[2]++
example 2:      MMOV32  MR1, *MAR1[-2]++
```

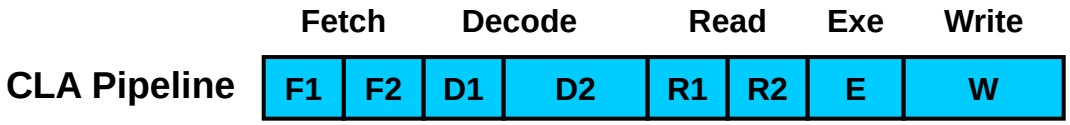
CLA Task Assembly Code Example

ClaTasks.asm

```
  .cdecls "Lab.h"
  .sect "Cla1Prog"
_Cla1Prog_Start:
_Cla1Task1:         ; FIR filter
      :
  MUI16TOF32 MR2, @_AdcResult.ADCRESULT0
  MMPYF32    MR2, MR1, MR0
      :
  MADDF32    MR3, MR3, MR2
  MF32TOUI16 MR2, MR3
  MMOV16     @_ClaFilteredOutput, MR2
      :
  MSTOP              ; End of task
;-----------------------------------
_Cla1Task2:
      :
  MSTOP
;-----------------------------------
_Cla1Task3:
      :
  MSTOP
```

- **.cdecls directive used to include the C header file in the CLA assembly file**
- **.sect directive used to place CLA assembly code in its own section**
- **C Peripheral Register Header File references can be used in CLA assembly code**
- **MSTOP instruction used at the end of the task**

CLA Pipeline

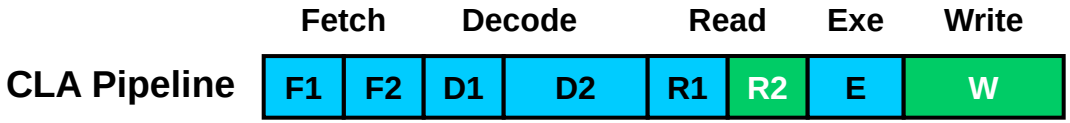


Independent 8 Stage Pipeline

Fetch1:	**Program read address generated**
Fetch2:	**Read Opcode via CLA program data bus**
Decode1:	**Decode instruction**
Decode2:	**Generate address**
	Conditional branch decision made
	MAR0/MAR1 update due to indirect addressing post increment
Read1:	**Data read address via CLA data read address bus**
Read2:	**Read data via CLA data read data bus**
Execute:	**Execute operation**
	MAR0/MAR1 update due to load operations
Write:	**Write**

All Instructions are single cycle (except for Branch/Call/Return)
Memory conflicts in F1, R1 and W stall the pipeline

Write Followed by Read




```
MMOV32  @_Reg1, MR3          ; Write Reg1
MMOV32  MR0, @_Reg2          ; Read Reg2
```

Due to the pipeline order, the read of Reg2 occurs before the Reg1 write

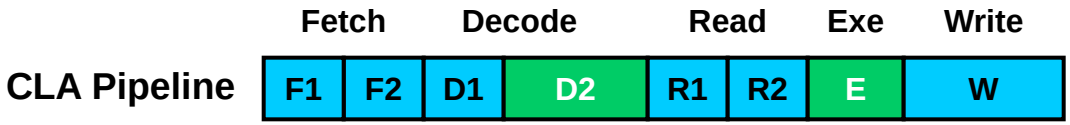
This is only an issue if the location written to can affect the location read
- Some peripheral registers
- Write to followed by read from the same location

Insert 3 other instructions or MNOPs to allow the write to occur first

Note: This behavior is different for the main C28 CPU:

The C28x CPU protects write followed by read to the same location
Blocks of peripheral registers have write-followed-by read protection

Loading MAR0 and MAR1



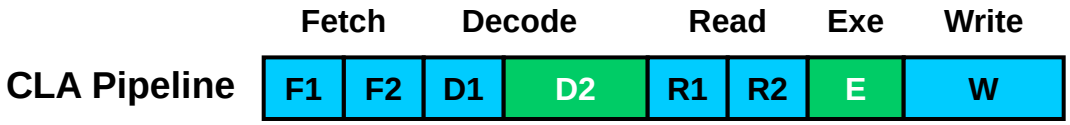
D2: Update to MAR0/MAR1 due to indirect addressing post increment
EXE: Update to MAR0/MAR1 due to load operation

Assume MAR0 is 50 and #_X is 20

```
MMOV16 MAR0, #_X          ; I1 Load MAR0 with 20
MMOV32 MAR1, *MAR0[0]++   ; I2 Uses old MAR0 Value (50)
MMOV32 MAR1, *MAR0[0]++   ; I3 Uses old MAR0 Value (50)
<Instruction 4>           ; I4 Can not use MAR0
MMOV32 MAR1, *MAR0[0]++   ; I5 Uses new MAR0 Value (20)
```

When instruction I1 is in EXE instruction I4 is in D2
If I4 uses MAR0, then a conflict will occur and MAR0 will not be loaded.

Branch, Call, Return Delayed Conditional



D2: Decide whether or not to branch
EXE: Branch taken (or not)

```
<Instruction 1>   ; I1 Last instruction to affect flags for branch

<Instruction 2>   ; I2
<Instruction 3>   ; I3   } Can not be branch or stop *
<Instruction 4>   ; I4   } Do not change flags in time to affect branch

 Branch, CND      ; MBCNDD, MCCNDD or MRCNDD

<Instruction 5>   ; I5
<Instruction 6>   ; I6   } Can not be branch or stop *
<Instruction 7>   ; I7   } Always executed whether branch is taken or not
```

* Can not be MSTOP (end of task), MDEBUGSTOP (debug halt), MBCNDD (branch), MCCNDD (call), or MRCNDD (return)

Optimizing Delayed Conditional Branch

6 instruction slots are executed on every branch

Use these slots to improve performance

MSTOP, MDEBUGSTOP MBCNDD, MCCNDD MRCNDD are not allowed in delay slots

```
        MCMPF32  MR0,#0.1
        MNOP
        MNOP
        MNOP
        MBCNDD   Skip1,NEQ
        MNOP
        MNOP
        MNOP
        MMOV32   MR1,@_Ramp
        MMOVXI   MR2,#RAMP_MASK
        MOR32    MR1,MR2
        MMOV32   @_Ramp,MR1
        ...
        MSTOP
Skip1:  MCMPF32  MR0,#0.01
        MNOP
        MNOP
        MNOP
        MBCNDD   Skip2,NEQ
        MNOP
        MNOP
        MNOP
        MMOV32   MR1,@_Coast
        MMOVXI   MR2,#COAST_MASK
        MOR32    MR1,MR2
        MMOV32   @_Coast,MR1
        ...
        MSTOP
Skip2:  MMOV32   MR3,@_Steady
        MMOVXI   MR2,#STEADY_MASK
        MOR32    MR3,MR2
        MMOV32   @_Steady,MR3
        ...
        MSTOP
```

Optimized Code

```
        MCMPF32  MR0,#0.1
        MCMPF32  MR0,#0.01
        MTESTTF  EQ
        MNOP
        MBCNDD   Skip1,NEQ
        MMOV32   MR1,@_Ramp
        MMOVXI   MR2,#RAMP_MASK
        MOR32    MR1,MR2
        MMOV32   @_Ramp,MR1
        ...
        MSTOP
Skip1:  MMOV32   MR3,@_Steady
        MMOVXI   MR2,#STEADY_MASK
        MOR32    MR3,MR2
        MBCNDD   Skip2,NTF
        MMOV32   MR1,@_Coast
        MMOVXI   MR2,#COAST_MASK
        MOR32    MR1,MR2
        MMOV32   @_Coast,MR1
        ...
        MSTOP
Skip2:  MMOV32   @_Steady,MR3
        ...
        MSTOP
```

CLA Initialization Code Example

Lab.h

```
#include "F2806x_Cla_typedefs.h"
#include "F2806x_Device.h"
                    ⋮
extern Uint16 Cla1Prog_Start;
extern interrupt void Cla1Task1();
extern interrupt void Cla1Task2();
                    ⋮
extern interrupt void Cla1Task8();
```

Cla.c

```
#include "Lab.h"


// Symbols used to calculate vector address
   Cla1Regs.MVECT1 =
         (Uint16)((Uint32)&Cla1Task1 -
                  (Uint32)&Cla1Prog_Start);
   Cla1Regs.MVECT2 =
         (Uint16)((Uint32)&Cla1Task2 -
                  (Uint32)&Cla1Prog_Start);
      ⋮
```

- **Defines data types and special registers specific to the CLA**
- **Defines register bit field structures**
- **Symbol for start of CLA program RAM defined in Lab.cmd**
- **CLA task prototypes are prefixed with the ‘interrupt’ keyword**
- **CLA task symbols are visible to all C28x CPU and CLA code**

MVECTx contains the offset address from the start of the CLA Program RAM

CLA Code Debugging

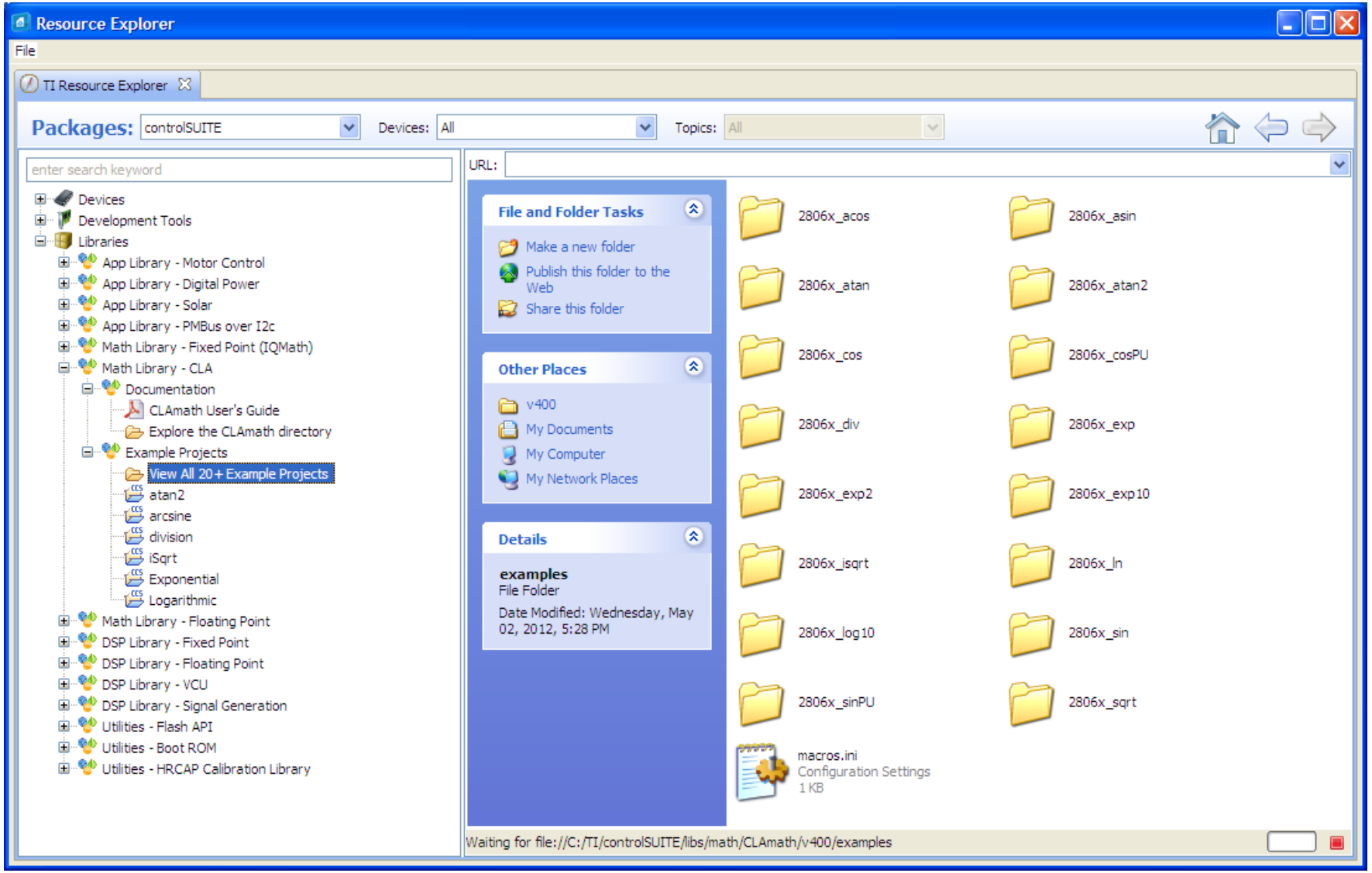
- ***The CLA can halt, single-step and run independently from the CPU***
- ***Both the CLA and CPU are debugged from the same JTAG port***

1. **Insert a breakpoint in CLA code**
   - Insert MDEBUGSTOP instruction to halt CLA and then rebuild/reload
2. **Enable CLA breakpoints**
   - Enable CLA breakpoints in the debugger
3. **Start the task**
   - Done by peripheral interrupt, software (IACK) or MIFRC register
   - CLA executes instructions until MDEBUGSTOP
   - MPC will the have address of MDEBUGSTOP instruction
4. **Single step the CLA code**
   - Once halted, single step the CLA code
   - Can also run to the next MDEBUGSTOP or to the end of task
   - If another task is pending it will start at end of previous task
5. **Disable CLA breakpoints, if desired**

Note: When debugging C code, the _mdebugstop() intrinsic places the MDEBUGSTOP instruction at that position in the generated assembly code

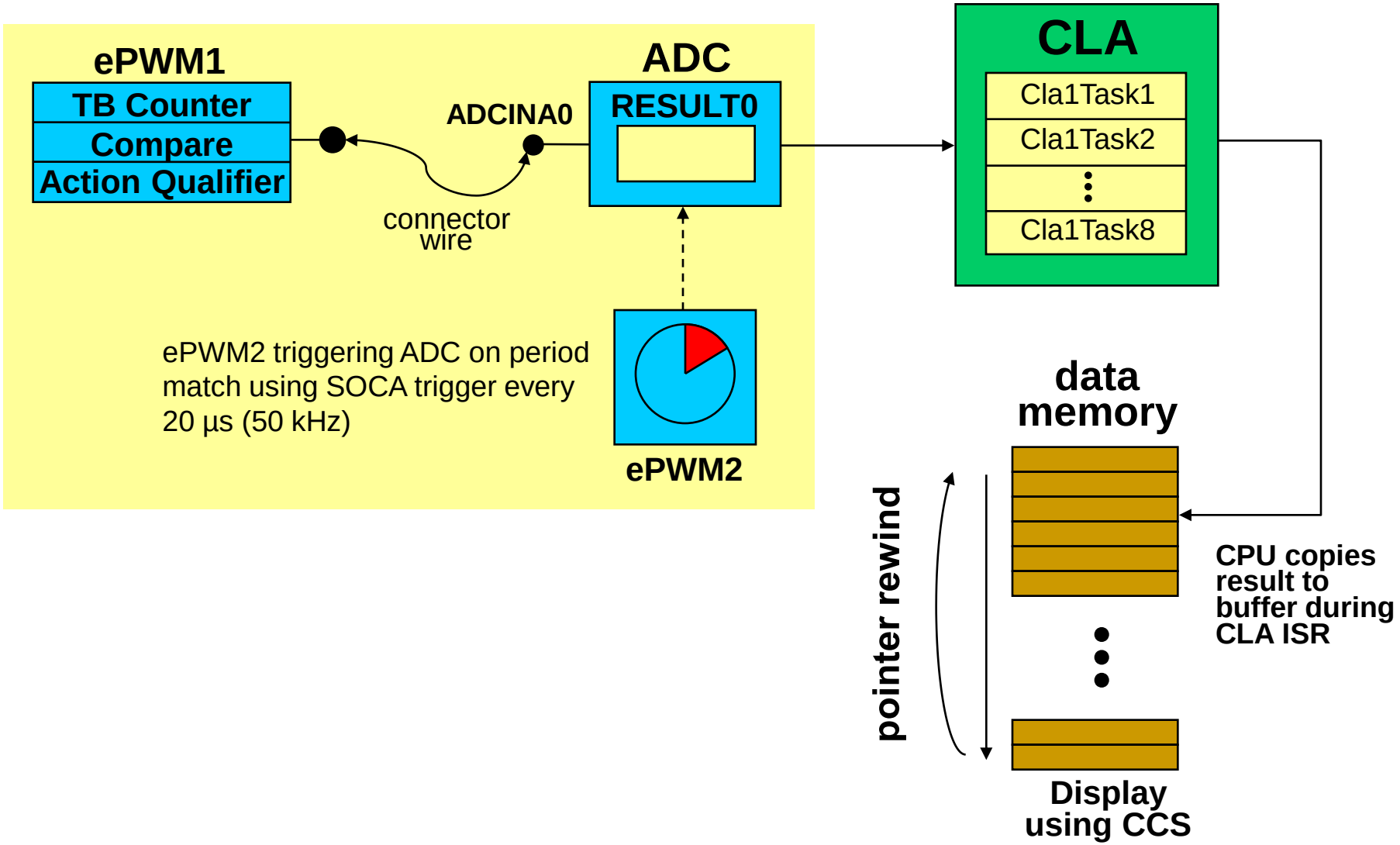
- *CLA single step – CLA pipeline is clocked only one cycle and then frozen*
- *CPU single step – CPU pipeline is flushed for each single step*

controlSUITE™ - CLA Software Support



- **TI provided functions to support floating-point math CLA operations**

Hands-on lab: CLA Floating-Point FIR Filter



F2806x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT9	XINT2	XINT1		ADCINT2	ADCINT1
INT2	EPWM8_TZINT	EPWM7_TZINT	EPWM6_TZINT	EPWM5_TZINT	EPWM4_TZINT	EPWM3_TZINT	EPWM2_TZINT	EPWM1_TZINT
INT3	EPWM8_INT	EPWM7_INT	EPWM6_INT	EPWM5_INT	EPWM4_INT	EPWM3_INT	EPWM2_INT	EPWM1_INT
INT4	HRCAP2_INT	HRCAP1_INT				ECAP3_INT	ECAP2_INT	ECAP1_INT
INT5				HRCAP4_INT	HRCAP3_INT		EQEP2_INT	EQEP1_INT
INT6			MXINTA	MRINTA	SPITXINTB	SPIRXINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8							I2CINT2A	I2CINT1A
INT9			ECAN1_INTA	ECAN0_INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10	ADCINT8	ADCINT7	ADCINT6	ADCINT5	ADCINT4	ADCINT3	ADCINT2	ADCINT1
INT11	CLA1_INT8	CLA1_INT7	CLA1_INT6	CLA1_INT5	CLA1_INT4	CLA1_INT3	CLA1_INT2	CLA1_INT1
INT12	LUF	LVF						XINT3

controlSTICK Header

F28069

1 ADC-A6 COMP3 (+VE)	2 ADC-A2 COMP1 (+VE)	3 ADC-A0	4 3V3
5 ADC-A4 COMP2 (+VE)	6 ADC-B1	7 EPWM-4B GPIO-07	8 TZ1 GPIO-12
9 SCL-A GPIO-33	10 ADC-B6 COMP3 (-VE)	11 EPWM-4A GPIO-06	12 ADC-A1
13 SDA-A GPIO-32	14 ADC-B0	15 EPWM-3B GPIO-05	16 5V0 (Disabled by Default)
17 EPWM-1A GPIO-00	18 ADC-B4 COMP2 (-VE)	19 EPWM-3A GPIO-04	20 SPISOMI-A GPIO-17
21 EPWM-1B GPIO-01	22 ADC-A5	23 EPWM-2B GPIO-03	24 SPISIMO-A GPIO-16
25 SPISTE-A GPIO-19	26 ADC-B2 COMP1 (-VE)	27 EPWM-2A GPIO-02	28 GND
29 SPICLK-A GPIO-18	30 GPIO-34 (LED)	31 PWM1A-DAC (Filtered)	32 GND

Edge of card