



TMS320C6x Architecture

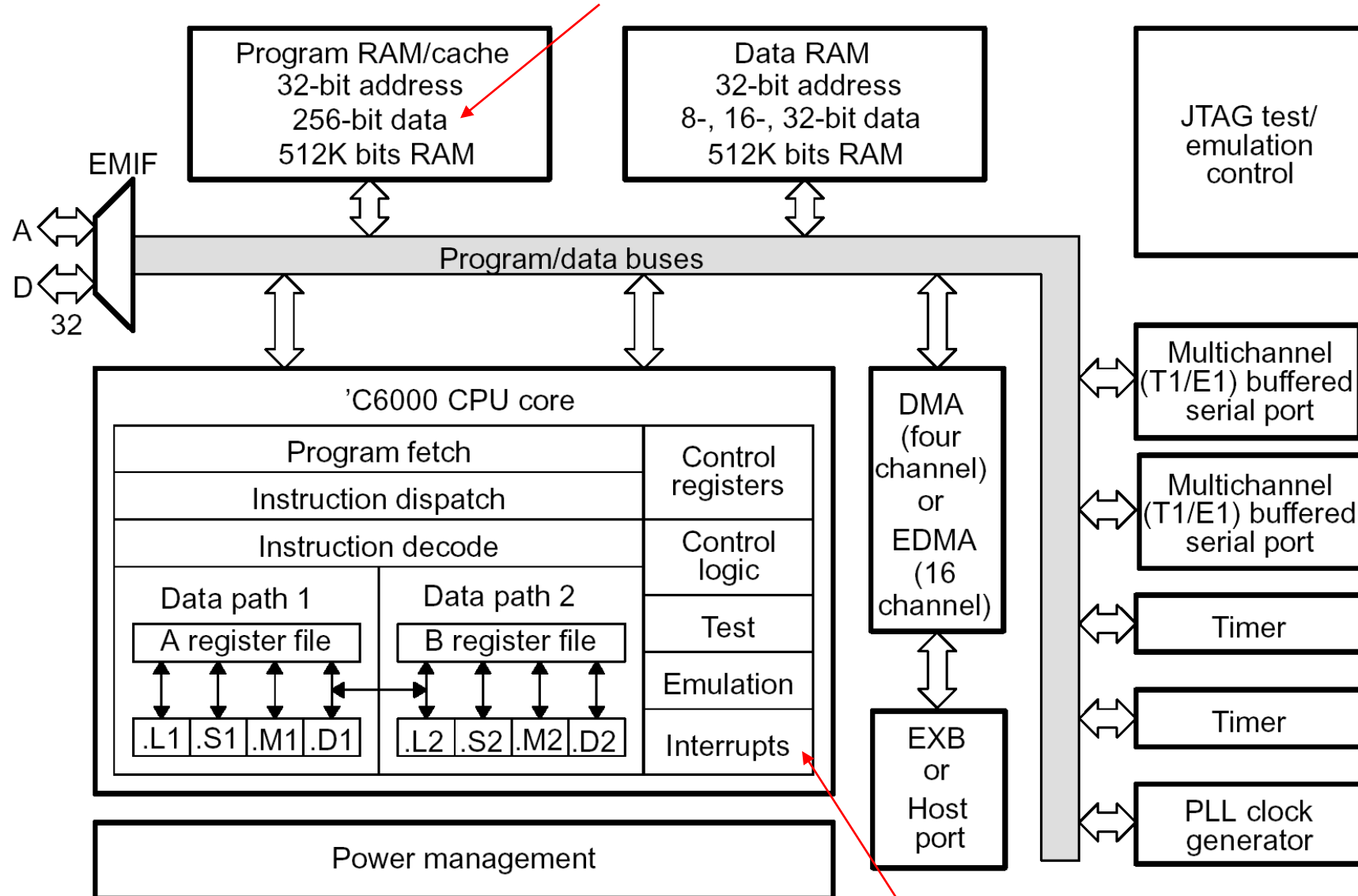
Hsiao-Lung Chan

Dept. Electrical Engineering

Chang Gung University

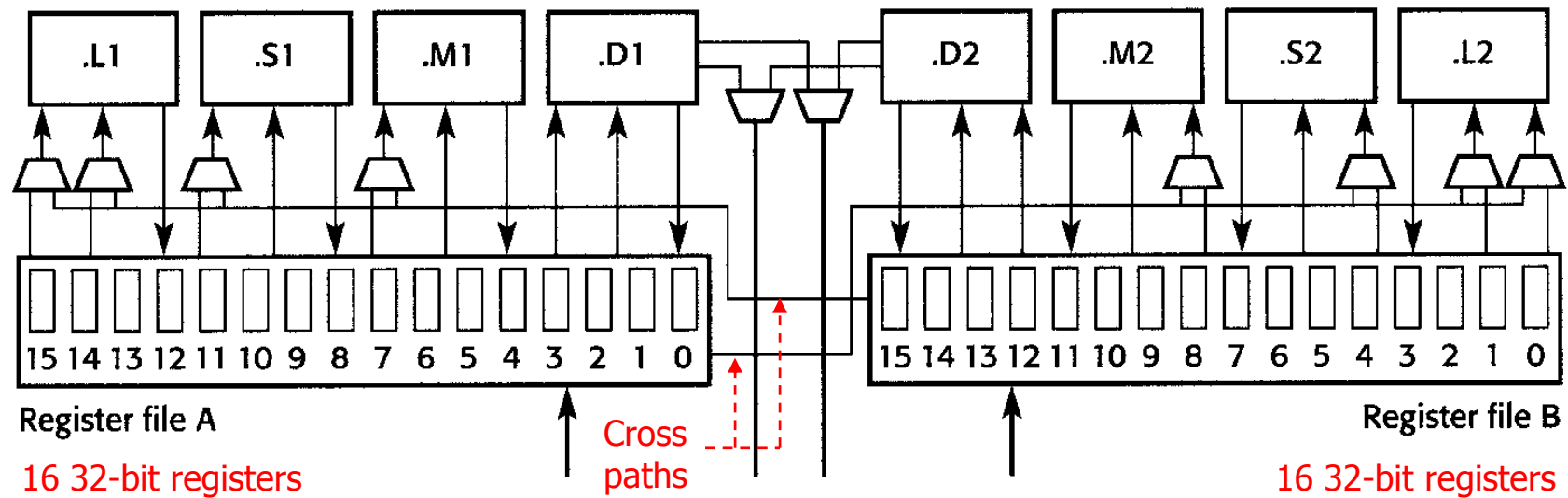
chanhl@mail.cgu.edu.tw

VLIW: Fetches eight 32-bit instructions every single cycle



14 interrupts: reset, NMI, INT4-INT15

Register Files



Register files support 32-, 40-
and 64-bit data formats

A1:A0	B1:B0
A3:A2	B3:B2
A5:A4	B5:B4
A7:A6	B7:B6
A9:A8	B9:B8
A11:A10	B11:B10
A13:A12	B13:B12
A15:A14	B15:B14

Up to 64 bits

Register File Example

MPY .M1 A1,A2,A3

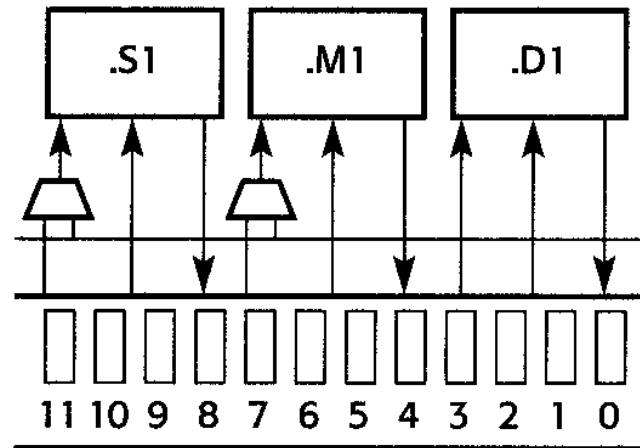


Table 2–1. Functional Units and Operations Performed

Functional Unit	Fixed-Point Operations	Floating-Point Operations
.L unit (.L1,.L2)	32/40-bit arithmetic and compare operations Leftmost 1 or 0 bit counting for 32 bits Normalization count for 32 and 40 bits 32-bit logical operations	Arithmetic operations Conversion operations: DP → SP, INT → DP, INT → SP
.S unit (.S1, .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from the control register file (.S2 only)	Compare reciprocal and reciprocal square-root operations Absolute value operations SP to DP conversion operations
.M unit (.M1, .M2)	16 × 16 bit multiply operations	32 × 32 bit multiply operations Floating-point multiply operations
.D unit (.D1, .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with a 5-bit constant offset Loads and stores with a 15-bit constant offset (.D2 only)	Load double word with a 5-bit constant offset

C64x/c67x Fixed-Point Instruction Set

.L Unit	.M Unit	.S Unit		.D Unit	
ABS	MPY	ADD	SET	ADD	STB (15-bit offset) [‡]
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bit offset) [‡]
ADDU	MPYUS	ADD2	SHR	ADDAH	STW (15-bit offset) [‡]
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SSHL	LDB	SUBAB
CMPGT	MPYHU	B IRP [†]	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRPT [†]	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB (15-bit offset) [‡]	
MV	MPYHULS	EXTU		LDBU (15-bit offset) [‡]	
NEG	MPYHSLU	MV		LDH (15-bit offset) [‡]	
NORM	MPYLH	MVC [†]		LDHU (15-bit offset) [‡]	
NOT	MPYLHU	MVK		LDW (15-bit offset) [‡]	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVKLH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

[†] S2 only

[‡] D2 only


Pipeline

- Fetch
- Decode
- Execute

Pipelined vs. Nonpipelined CPU

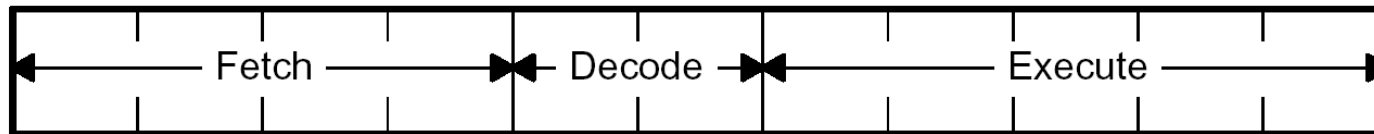
CPU Type	Clock Cycles								
	1	2	3	4	5	6	7	8	9
Non-Pipelined	F ₁ D ₁ E ₁			F ₂ D ₂ E ₂			F ₃ D ₃ E ₃		
Pipelined	F ₁	D ₁ F ₂	E ₁ D ₂ F ₃	E ₂ D ₃	E ₃				

F_x = fetching of instruction x
 D_x = decoding of instruction x
 E_x = execution of instruction x

 Pipeline full

Pipeline Operation of C6x DSP

- C62x and C64x

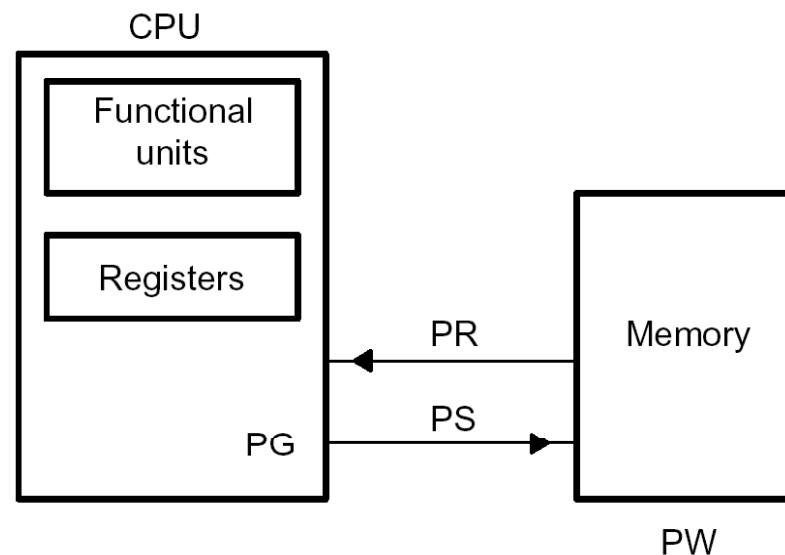


- C67x



Fetch Phase

- Fetching consists of 4 phases, each requiring a clock cycle
 - PG: Program address generation
 - PS: Program address send to memory
 - PW: Program address ready wait (memory read occur)
 - PR: Program fetch package receive at CPU
- (256-bit VLIW enables 8 instructions/package)



Decode Phase

- Decoding consists of 2 phases, each requiring a clock cycle
 - DP: Instruction dispatch to appropriate functional units
 - DC: Instruction decode

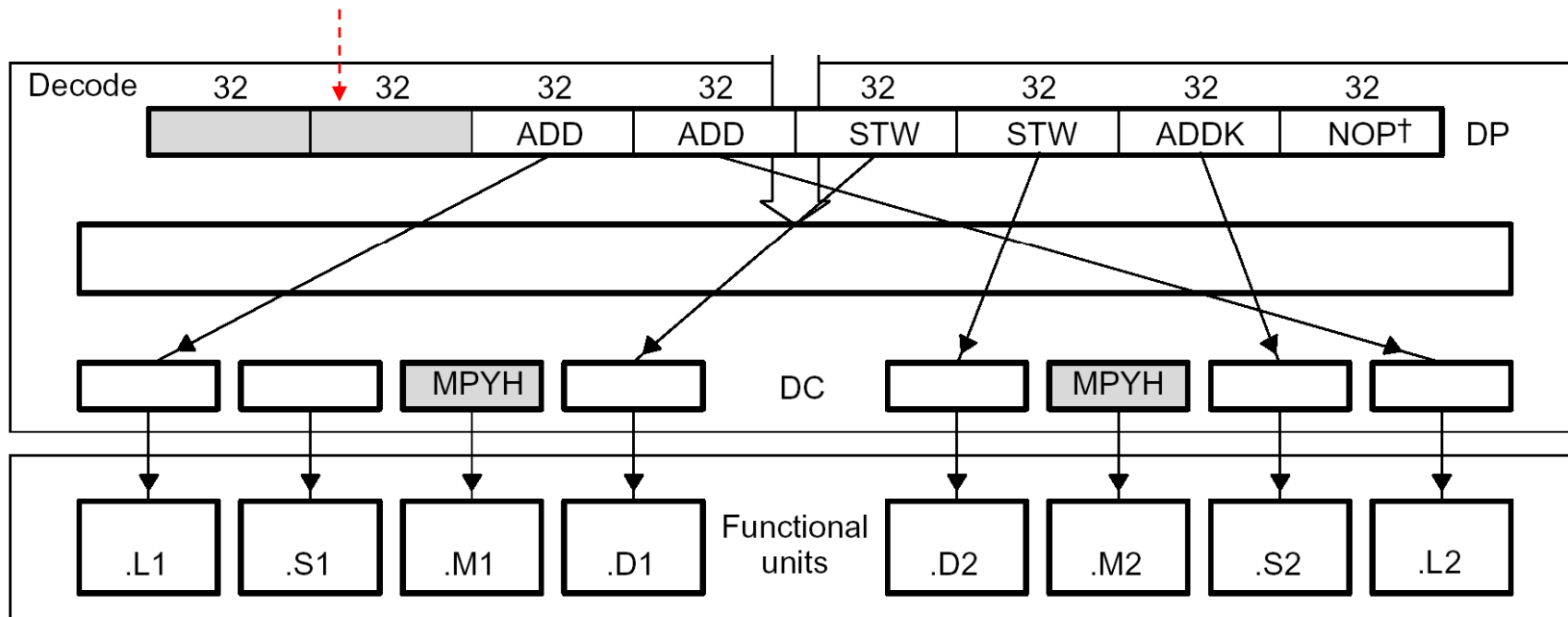
Decode Phase (Cont.)

```

MPYH .M1 op
|| MPYH .M2 op

ADD .L1 op
|| ADD .L2 op
|| STW .D1 op
|| STW .D2 op
|| ADDK .S2 op
|| NOP
    
```

The packet of two parallel instructions that were dispatched on the previous cycle



† NOP is not dispatched to a functional unit.

Execute Phase

- Pipeline of fixed-point DSP (C62x, C64x) has 5 phases:
E1 – E5
- Pipeline of floating-point DSP (C67x) has 10 phases:
E1 – E10
- Different types of instructions require different numbers of executing phases to complete their execution
- Delay slots
 - Occupy CPU cycles after E1 of an instruction
 - Subsequent instructions are not available until the the end of the last delay slot

Execution Stage Length of C62x and C64x

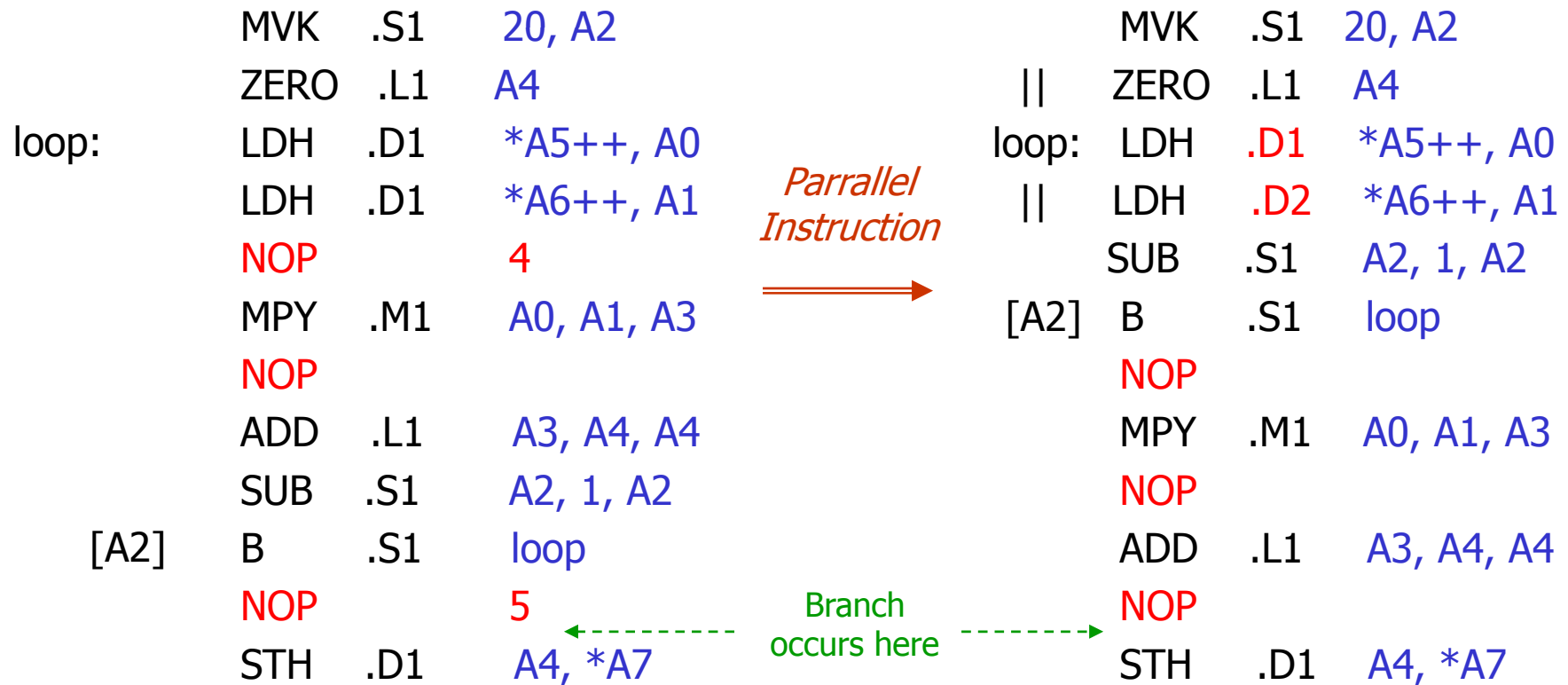
		Instruction Type					
		Single Cycle	16 X 16 Single Multiply/ C64x .M Unit Non-Multiply	Store	C64x Multiply Extensions	Load	Branch
Execution phases	E1	Compute result and write to register	Read operands and start computations	Compute address	Reads operands and start computations	Compute address	Target-code in PG‡
	E2		Compute result and write to register	Send address and data to memory		Send address to memory	
	E3			Access memory		Access memory	
	E4				Write results to register	Send data back to CPU	Branch occurs after E5
	E5					Write data into register	
Delay slots		0	1	0†	3	4†	5‡

Dot Product Example: $y = \sum_{n=1}^{20} a(n) * x(n)$

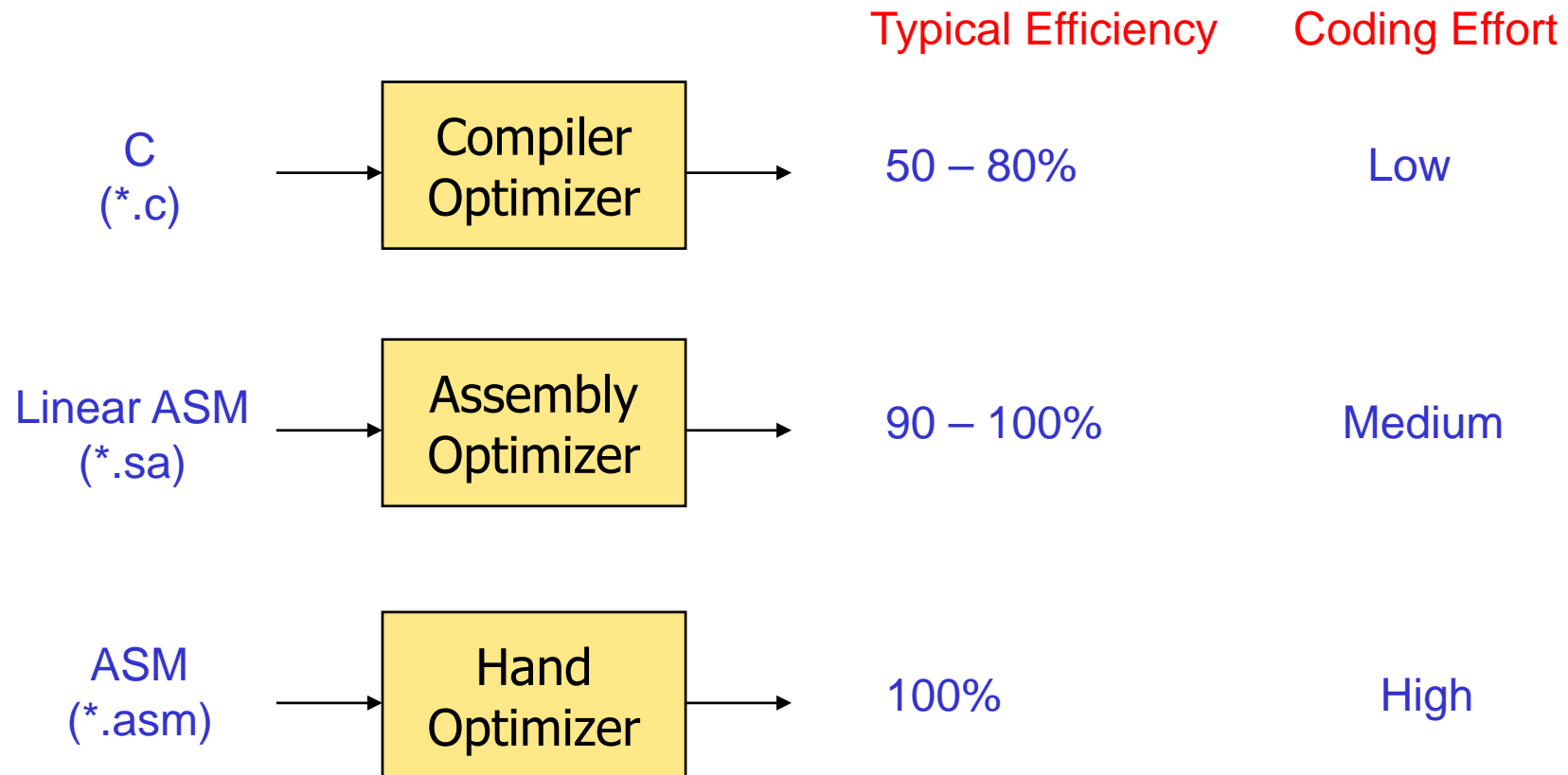
<u>Label</u>	<u>Instruction</u>	<u>Operands</u>	<u>Comment</u>
	MVK .S1	a, A5	; Get address of a
	MVKH .S1	a, A5	
	MVK .S1	x, A6	; Get address of x
	MVKH .S1	x, A6	
	MVK .S1	y, A7	; Get address of y
	MVKH .S1	y, A7	
	MVK .S1	20, A2	; Set loop counter, A2=20
	ZERO .L1	A4	; A4 = 0
	LDH .D1	*A5++, A0	; A0 = a (n)
loop:	LDH .D1	*A6++, A1	; A1 = x(n)
	NOP 4		
	MPY .M1	A0, A1, A3	; A3 = a(n) * x(n)
	NOP		<i>No operation but occupying a cycle</i>
	ADD .L1	A3, A4, A4	; A4 = A4 + a(n) * x(n)
	SUB .S1	A2, 1, A2	; Decrement loop counter
[A2]	B .S1	loop	; if A2 ≠ 0, branch to loop
	NOP 5		
	STH .D1	A4, *A7	; y = A4

Function unit  Data path: 1 for path A, and 2 for path B

Considering NOP and Parallelism



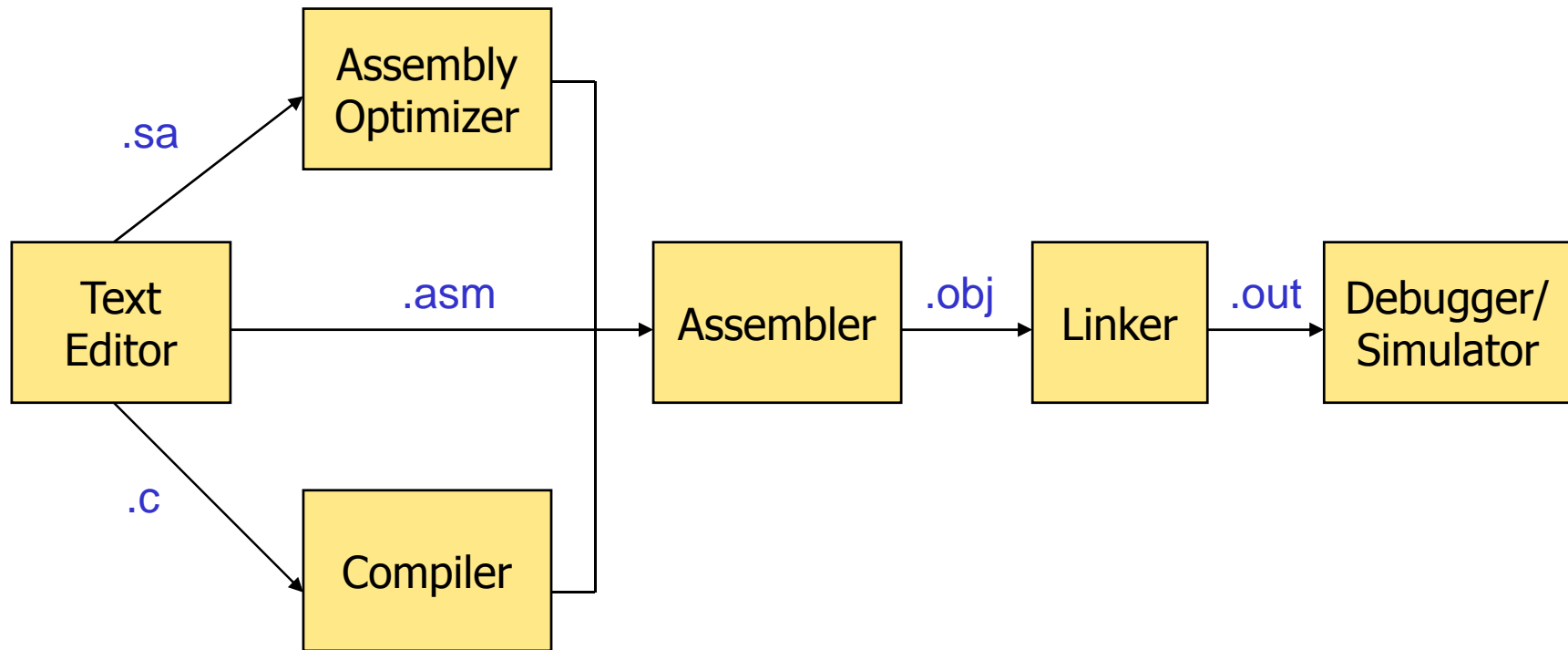
Code Efficiency vs. Coding Effort



Linear Assembly Code

- Assembly optimizer take care of :
 - Finding the instruction can be executed in parallel
 - Handling pipeline latencies
 - Assign register usage
 - Define which functional unit to be used

C6x Software Tools



Dot Product: C

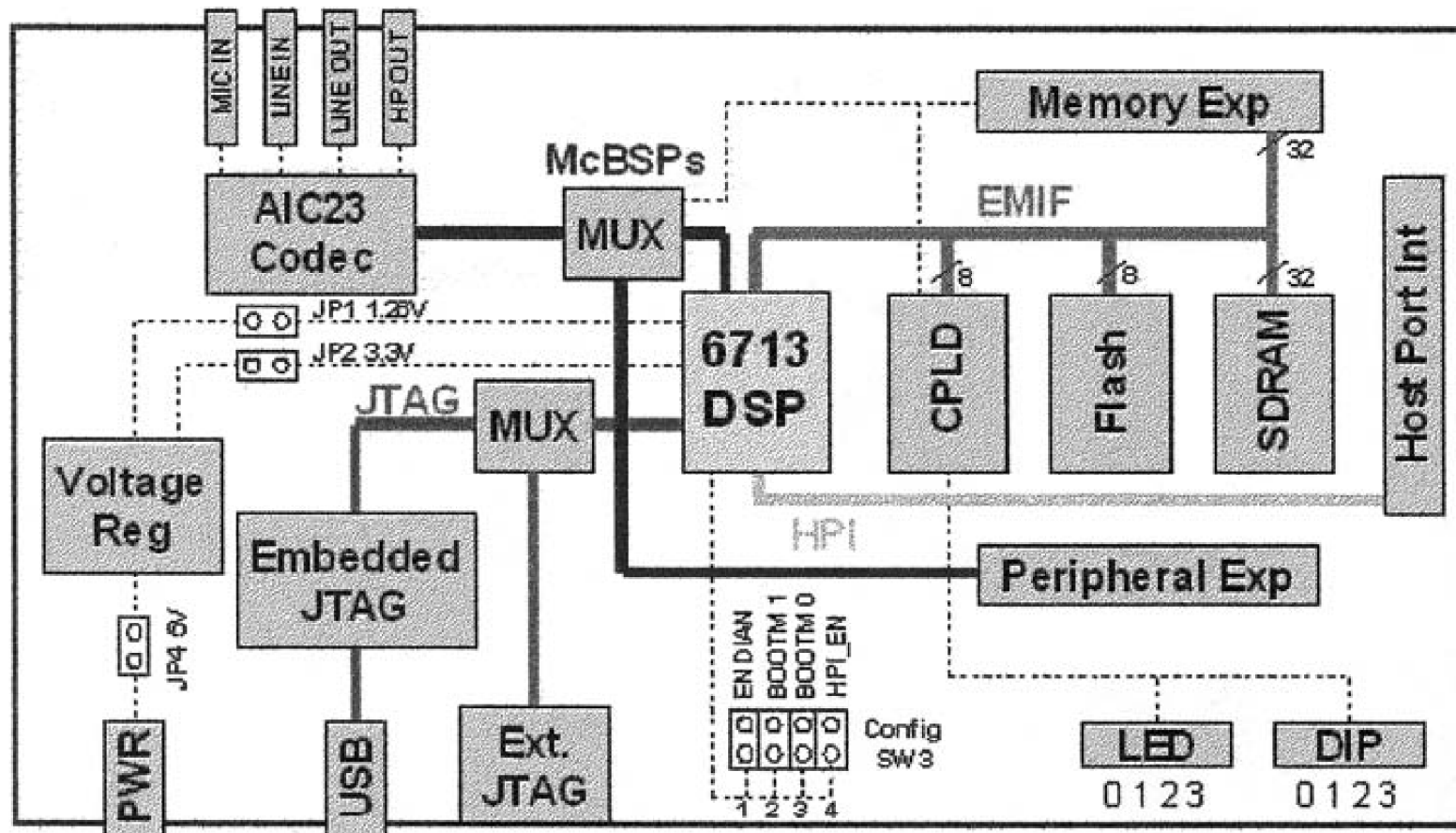
```
int dotp(int *a, int *b, short cnt)
{
    int sum, i;
    sum=0;
    for (i=0; i<cnt; i++)
        sum += a[i] * b[i];
    return(sum);
}
```

Dot Product: Linear Assembly

```
dotp: .cproc  ptr_a, ptr_b, cnt
      .reg    val1, val2, prod, sum
      zero    sum
loop: ldh     *ptr_a++, val1
      ldh     *ptr_b++, val2
      mpy     val1, val2, prod
      add     sum, prod, sum
      sub     cnt, 1, cnt
[cnt] b      loop
      return  sum
      .endproc
```

C6713 DSK

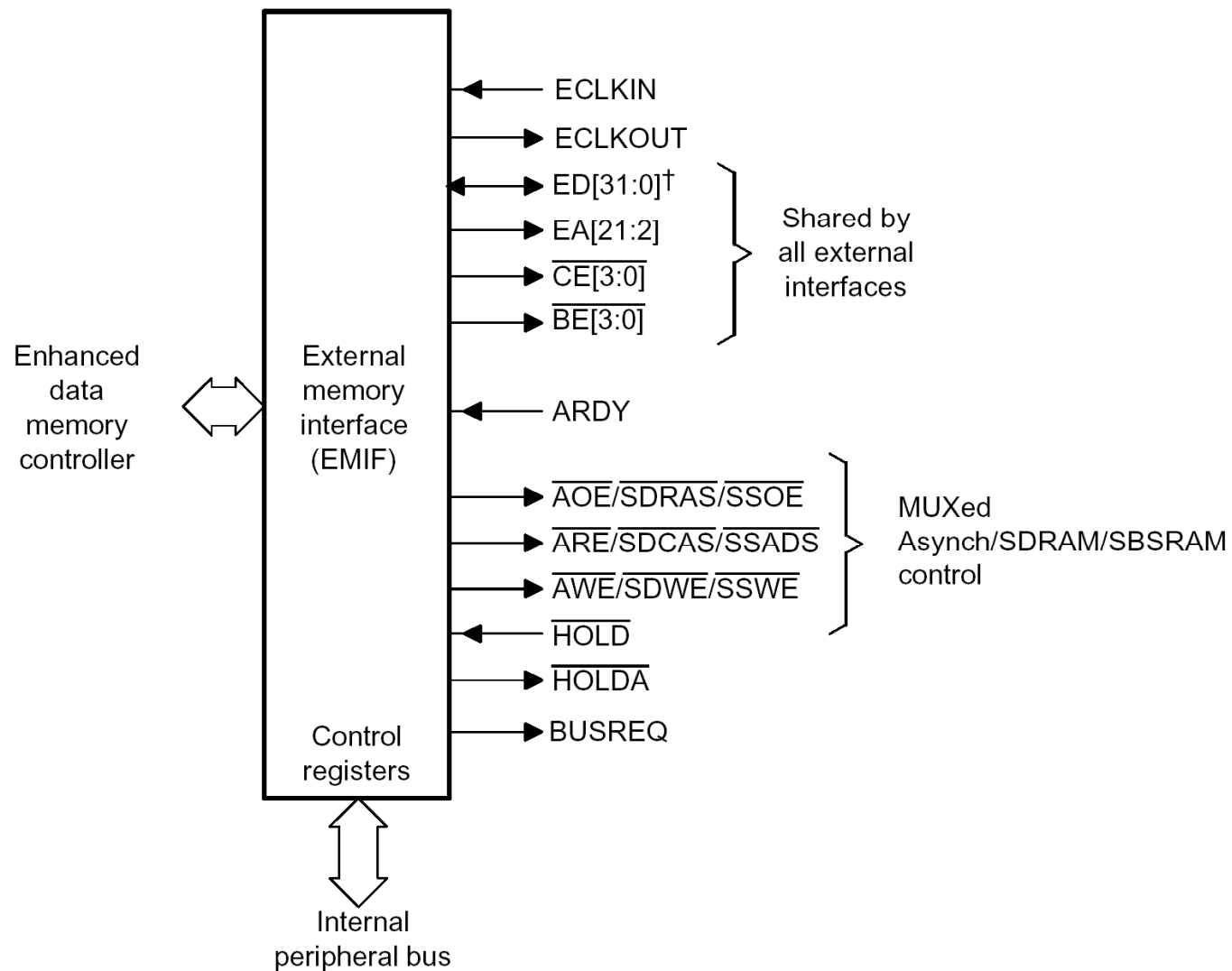
24-bit stereo codec with up to 48 kHz sampling rate



Memory Map

- 4 Gbytes (2^{32}) memory space
 - Internal program memory
 - Internal data memory
 - External memory spaces
 - Internal peripheral space

External Memory Interface (EMIF)

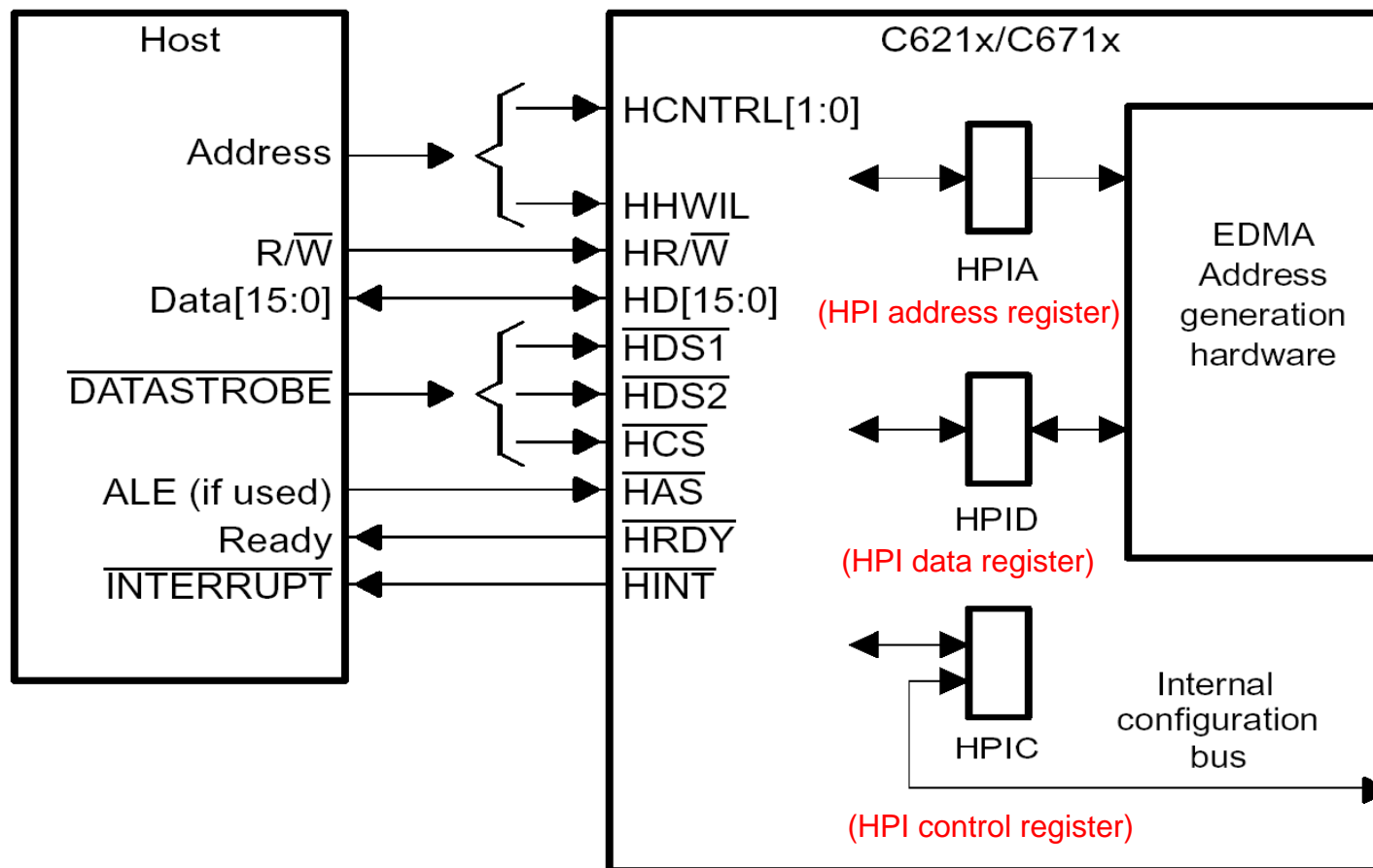


Multichannel Buffer Serial Port (McBSP)

- Full-duplex communication
- Double-buffered data registers
- Independent framing and clocking for receive and transmit
- Direct interface to codecs, analog interface chips (AICs), and other serially connected analog-to-digital (A/D) and digital-to-analog (D/A) chips

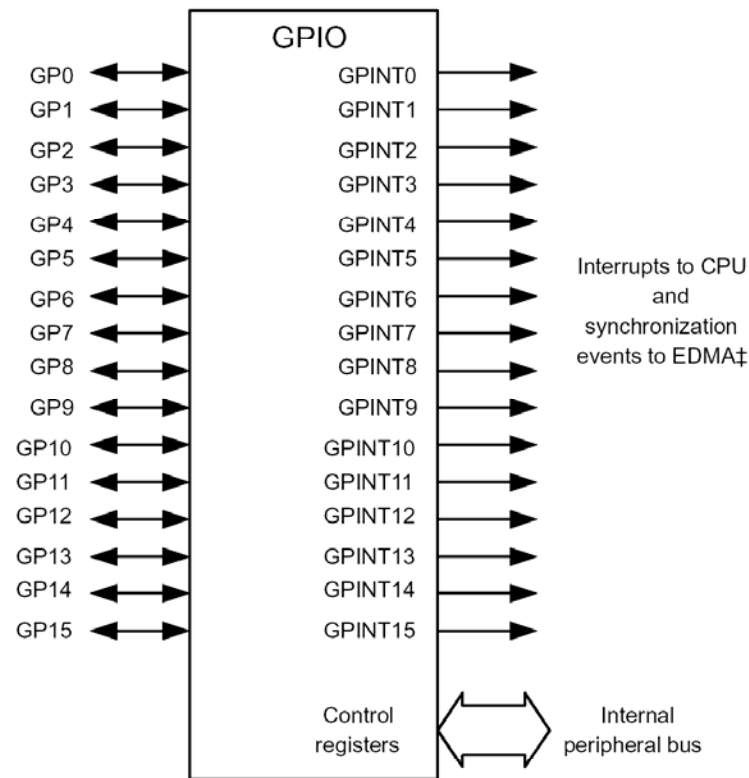
Host Peripheral Interface (HPI)

- A parallel port through which a host processor can directly access the CPU's memory space



General-Purpose Interrupt I/O (GPIO)

- Can be configured as either inputs or outputs
- GPIO can produce CPU interrupts and EDMA events



† Some of the GPx pins are MUXed with other device signals. Refer to the specific device datasheet for details.

‡ All GPINTx are synchronization events to the EDMA. Only GPINT0 and GPINT[4:7] are available as interrupts to the CPU.

Enhanced Direct Memory Access (EDMA)

- Transfer data between regions in the **memory map** without CPU intervention (background CPU operation)
 - Internal memory, internal peripherals, or external devices

Interrupts

- 16 interrupt sources
 - Host port to DSP interrupt (DSPINT)
 - Timer interrupt (TINT0, TINT1)
 - EMIF SDRAM timer interrupt (SD_INT)
 - External interrupts (EXT_INT4 ~7)
 - DMA interrupts (DMA_INT0~3)
 - McSBP transmit/receiver interrupts (XINT0~1/RINT0~1)

Interrupt Priority

Priority	Interrupt Name
Highest	Reset
	NMI ← Nonmaskable Interrupt: Alert the CPU of a serious hardware problem such as imminent power failure.
	INT4
	INT5
	INT6
	INT7
	INT8
	INT9
	INT10
	INT11
	INT12
	INT13
	INT14
Lowest	INT15

Maskable Interrupt:
Associated with external devices, on-chip peripherals, software control

Interrupt Service Table (IST)

Interrupt Service Fetch Packet (ISFP)

- Contain code for servicing the interrupts.
- Each with eight 32-bit instruction words (32 bytes)

ISFP for INT6

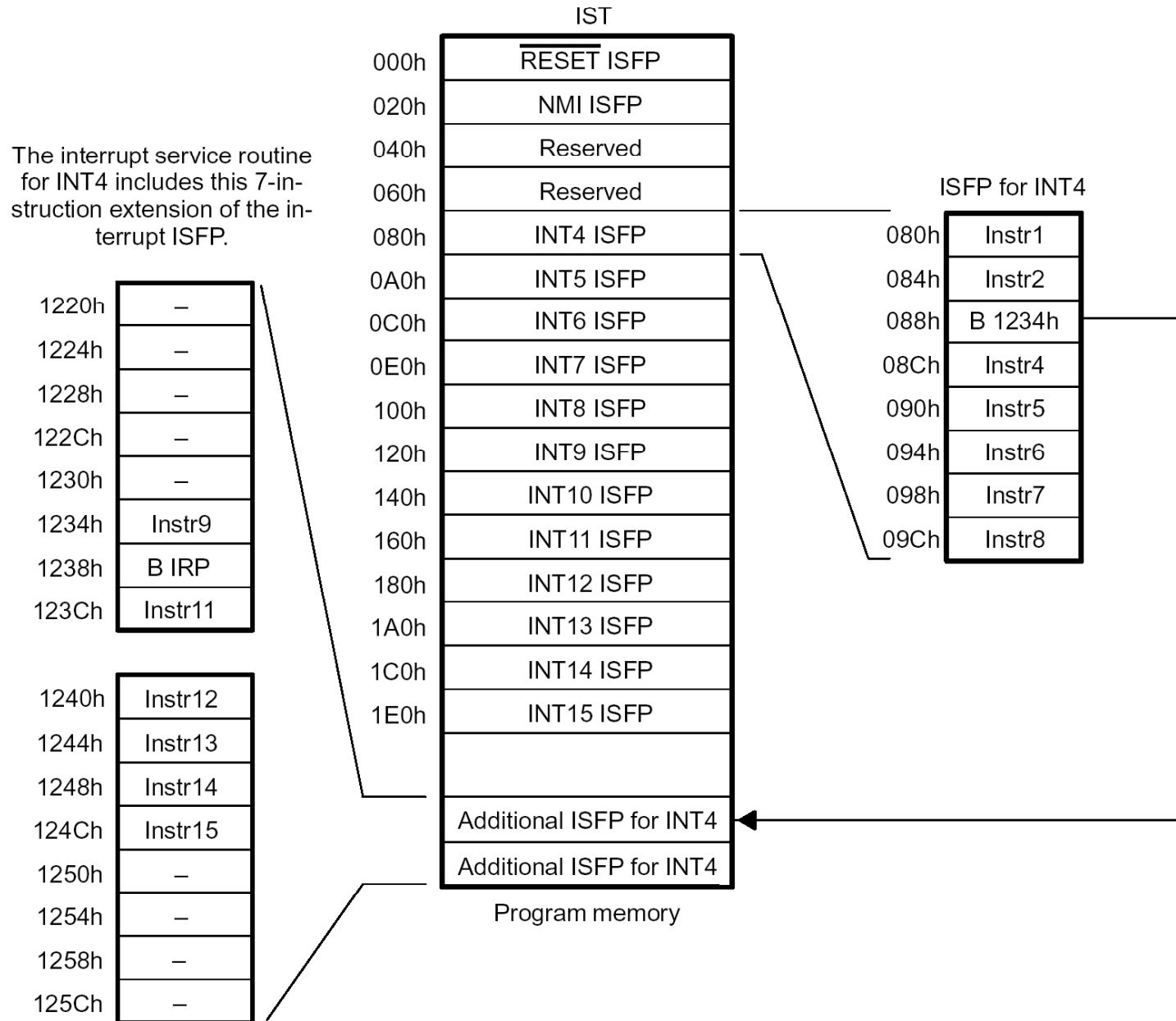
0C0h	Instr1
0C4h	Instr2
0C8h	Instr3
0CCh	Instr4
0D0h	Instr5
0D4h	Instr6
0D8h	B IRP
0DCh	NOP 5

The interrupt service routine for INT6 is short enough to be contained in a single fetch packet.

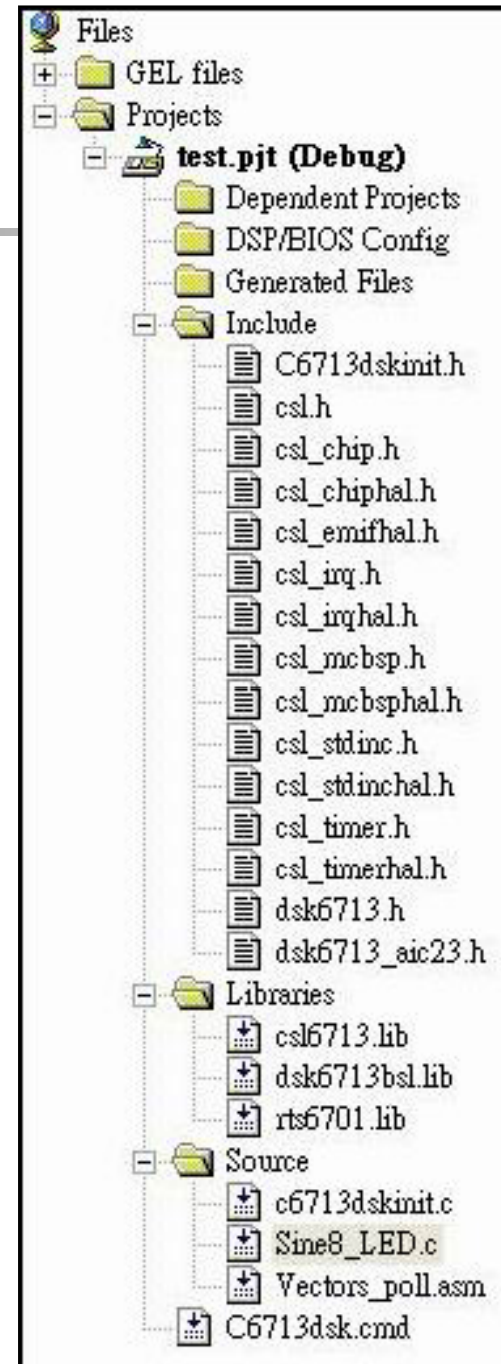
000h	RESET ISFP
020h	NMI ISFP
040h	Reserved
060h	Reserved
080h	INT4 ISFP
0A0h	INT5 ISFP
0C0h	INT6 ISFP
0E0h	INT7 ISFP
100h	INT8 ISFP
120h	INT9 ISFP
140h	INT10 ISFP
160h	INT11 ISFP
180h	INT12 ISFP
1A0h	INT13 ISFP
1C0h	INT14 ISFP
1E0h	INT15 ISFP

Program memory

IST with Branch to Additional Interrupt Service Code Located outside the IST



Project view for 6713 DSK



C6713dsk.cmd

MEMORY

```
{  
  IVECS:          org=0h,          len=0x220  
  IRAM:   org=0x00000220, len=0x0002FDE0 /*internal memory*/  
  SDRAM:  org=0x80000000, len=0x00100000 /*external memory*/  
  FLASH:  org=0x90000000, len=0x00020000 /*flash memory*/  
}
```

SECTIONS

```
{  
  .EXT_RAM :> SDRAM  
  .vectors :> IVECS /*in vector file*/  
  .text    :> IRAM  /*Created by C Compiler*/  
  .bss     :> IRAM  
  .cinit   :> IRAM  
  .stack   :> IRAM  
  .sysmem  :> IRAM  
  .const   :> IRAM  
  .switch  :> IRAM  
  .far     :> IRAM  
  .cio     :> IRAM  
  .csldata :> IRAM  
}
```

C6713dskinit.h and C6713dskinit.c

```
void c6713_dsk_init();  
void comm_poll();  
void comm_intr();  
Uint32 input_sample();  
short input_left_sample();  
short input_right_sample();  
void output_sample(int);  
void output_left_sample(short);  
void output_right_sample(short);
```

dsk6713_aic23.h

```
#define DSK6713_AIC23_FREQ_8KHZ      1
#define DSK6713_AIC23_FREQ_16KHZ     2
#define DSK6713_AIC23_FREQ_24KHZ     3
#define DSK6713_AIC23_FREQ_32KHZ     4
#define DSK6713_AIC23_FREQ_44KHZ     5
#define DSK6713_AIC23_FREQ_48KHZ     6
#define DSK6713_AIC23_FREQ_96KHZ     7
```

Example: Sine8_buf

```
#include "dsk6713_aic23.h"           //support file for codec,DSK
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
int loop = 0;
short gain = 10;
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};
short out_buffer[256];
const short BUFFERLENGTH = 256;
int i = 0;

interrupt void c_int11()             //interrupt service routine
{
    output_sample(sine_table[loop]*gain); //output sine values
    out_buffer[i] = sine_table[loop]*gain; //output to buffer
    i++;
    if(i==BUFFERLENGTH) i=0;
    if (loop < 7) ++loop;
    else loop = 0;
    return;
}

void main()
{
    comm_intr();           //init DSK, codec, McBSP
    while(1);              //infinite loop
}
```

Example: Sine_stereo

```
#include "dsk6713_aic23.h"           //codec-dsk support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate

#define LEFT 0
#define RIGHT 1
union {Uint32 combo; short channel[2];} AIC23_data;

short loop = 0, gain = 10;
short sine_table[8] = {0,707,1000,707,0,-707,-1000,-707};

interrupt void c_int11()//interrupt service routine
{
    AIC23_data.channel[RIGHT]=sine_table[loop]*gain; //for right channel;
    AIC23_data.channel[LEFT]=sine_table[loop]*gain;  //for leftchannel;
    output_sample(AIC23_data.combo); //output to both channels
    if (++loop > 7) loop = 0;
}

void main()
{
    comm_intr();           //init DSK,codec,McBSP
    while(1) ;             //infinite loop
}
```

Example: Sine8_LED

```
#include "dsk6713_aic23.h"           //support file for codec,DSK
Uint32 fs = DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
short    loop = 0;
short gain = 10;
short sine_table[8]={0,707,1000,707,0,-707,-1000,-707};

void main()
{
    comm_poll();                     //init DSK,codec,McBSP
    DSK6713_LED_init();               //init LED from BSL
    DSK6713_DIP_init();               //init DIP from BSL
    while(1)
    {
        if(DSK6713_DIP_get(0)==0)    //==0 if DIP switch #0 pressed
        {
            DSK6713_LED_on(0);        //turn LED #0 ON
            output_sample(sine_table[loop]*gain);
            if (loop < 7) ++loop;
            else loop = 0;
        }
        else DSK6713_LED_off(0);      //turn LED off if not pressed
    }
}
```

Example: Sine2sliders

```
#include "DSK6713_AIC23.h"           //codec-DSK interface support
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  //set sampling rate

short loop = 0;
short sine_table[32]={0,195,383,556,707,831,924,981,1000,981,924,831,707,556,383,195,
    0,-195,-383,-556,-707,-831,-924,-981,-1000, -981,-924,-831,-707,-556,-383,-195};
short gain = 1;                      //for slider
short frequency = 2;                 //for slider

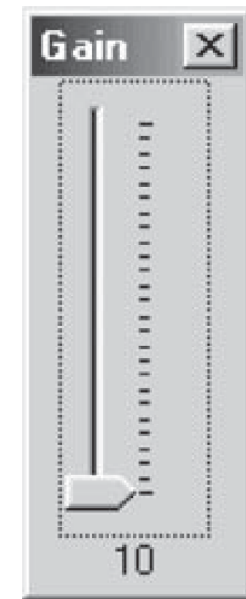
void main()
{
    comm_poll();                     //init DSK,codec,McBSP
    while(1)
    {
        output_sample(sine_table[loop]*gain);
        loop += frequency;           //increase frequency index
        loop = loop % 32;
    }
}
```


Slider Gel

Applying the Slider Gel File

The General Extension Language (GEL) is an interpretive language similar to (a subset of) C. It allows you to change a variable such as gain, sliding through different values while the processor is running. All variables must first be defined in your source program.

1. Select File → Load GEL and open the file `gain.gel`, which you retained from the original folder, `sine8_LED` (that you backed up). Double-click on the file `gain.gel` to view it within CCS. It should be displayed in the Files window. This file is shown in Figure 1.6. By creating the slider function `gain` shown in Figure 1.6, you can start with an initial value of 10 (first value) for the variable `gain` that is set in the C program, up to a value of 35 (second value), incremented by 5 (third value).
2. Select GEL → Sine Gain → Gain. This should bring out the Slider window shown in Figure 1.7, with the minimum value of 10 set for the gain.
3. Press the up-arrow key to increase the gain value from 10 to 15, as displayed in the Slider window. Verify that the volume of the sine wave generated has increased. Press the up-arrow key again to continue increasing the slider, incrementing by 5 up to 30. The amplitude of the sine wave should be about 2.5 V p-p with a `gain` value set at 30. Now use the mouse to click directly on the Slider window and slowly increase the slider position to 31, then 32, and



Example: Sinegen_table

```
#include "DSK6713_AIC23.h"           //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  //set sampling rate
#include <math.h>
#define table_size (short)10
short sine_table[table_size];
int i;

interrupt void c_int11()    //interrupt service routine
{
    output_sample(sine_table[i]);
    if (i < table_size - 1) ++i;
    else i = 0;
    return;
}

void main()
{
    float pi=3.14159;
    for(i = 0; i < table_size; i++)
        sine_table[i]=10000*sin(2.0*pi*i/table_size);
    i = 0;
    comm_intr();           //init DSK, codec, McBSP
    while(1);
}
```

Example: Loop_intr

//Loop program using interrupt. Output = delayed input

```
#include "dsk6713_aic23.h"           //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
```

```
interrupt void c_int11()           //interrupt service routine
{
    short sample_data;

    sample_data = input_sample(); //input data
    output_sample(sample_data);   //output data
    return;
}
```

```
void main()
{
    comm_intr();                   //init DSK, codec, McBSP
    while(1);
}
```

Example: Loop_stereo

```
// Stereo input/output to/from both channels
#include "dsk6713_aic23.h"           //codec-DSK support file
Uint32 fs=DSK6713_AIC23_FREQ_8KHZ; //set sampling rate
#define LEFT    0
#define RIGHT   1
union {Uint32 combo; short channel[2];} AIC23_data;

interrupt void c_int11()           //interrupt service routine
{
    AIC23_data.combo = input_sample();           //input 32-bit sample
    output_left_sample(AIC23_data.channel[LEFT]); //I/O left channels
    return;
}

void main()
{
    comm_intr();                               //init DSK, codec, McBSP
    while(1);
}
```

Reference

- A. Bateman, The DSP Handbook: Algorithms, Applications, and Design Techniques, Prentice-Hall, 2002
- Rulph Chassaing, DSP Applications Using C and the TMS320C6x DSK, John Wiley & Sons Inc., 2002.
- N. Kehtarnavaz, B. Simsek, C6x-Based Digital Signal Processing, Prentice-Hall, 2000.
- TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instrument, 2000.
- TMS320C6000 Programmer's Guide, Texas Instrument, 2001.