Lecture 4

TMS320C6711 DSK Architecture

- Lab 3 due one week from today (shortest lab)

Assembly Language Programming

- It is possible to have a project that is coded entirely in assembly language. (see the textbook for an example)

- In this course, we will only consider assembly coded functions that are called from a C program (i.e. from the main() program (as in Lab 3) or from an interrupt as in Lab 4 and other real-time applications that use interrupts)

- In Lab 3, real-time processing will not be done. Therefore, the codec does not need to be initialized (i.e. the commands comm_poll() or comm_intr() will not appear in the main() function and the files C6xdskinit.c & .h will not appear in the project file).

- Assembly language is required for memory critical & speed critical parts of algorithms. For this class, we will use it to gain insight as to how algorithms are processed on the chip.

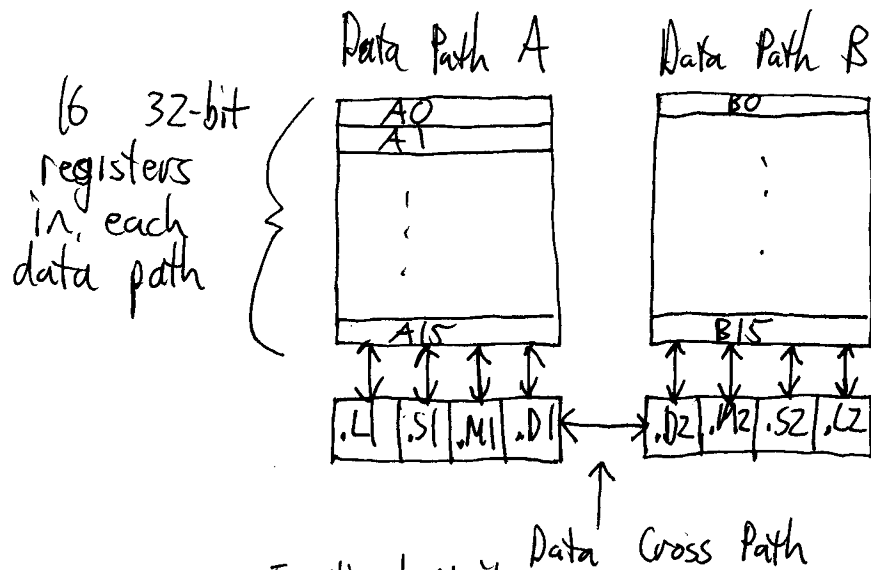
- Assembly language programming requires the programmer to break down an algorithm into chronologically organized operations and assign these operations to registers and ALU's (more to come on this in a moment).

~ SHOW SLIDE ~

TMS320C6711 DSP chip

- DSP Core — TI Doc SPRU189f
- Memory, Synchronization, External Device Interfaces — TI Doc SPRU190d

DSP Core or Central Processing Unit (CPU) (Abridged)



TI Doc Notation

word – 32 bits (float / int)

half-word – 16 bits (short)

1 ⇔ data path A

2 ⇔ data path B

ALU ≡ arithmetic logic unit

Functional Units

.L : fixed & floating-pt. ALU

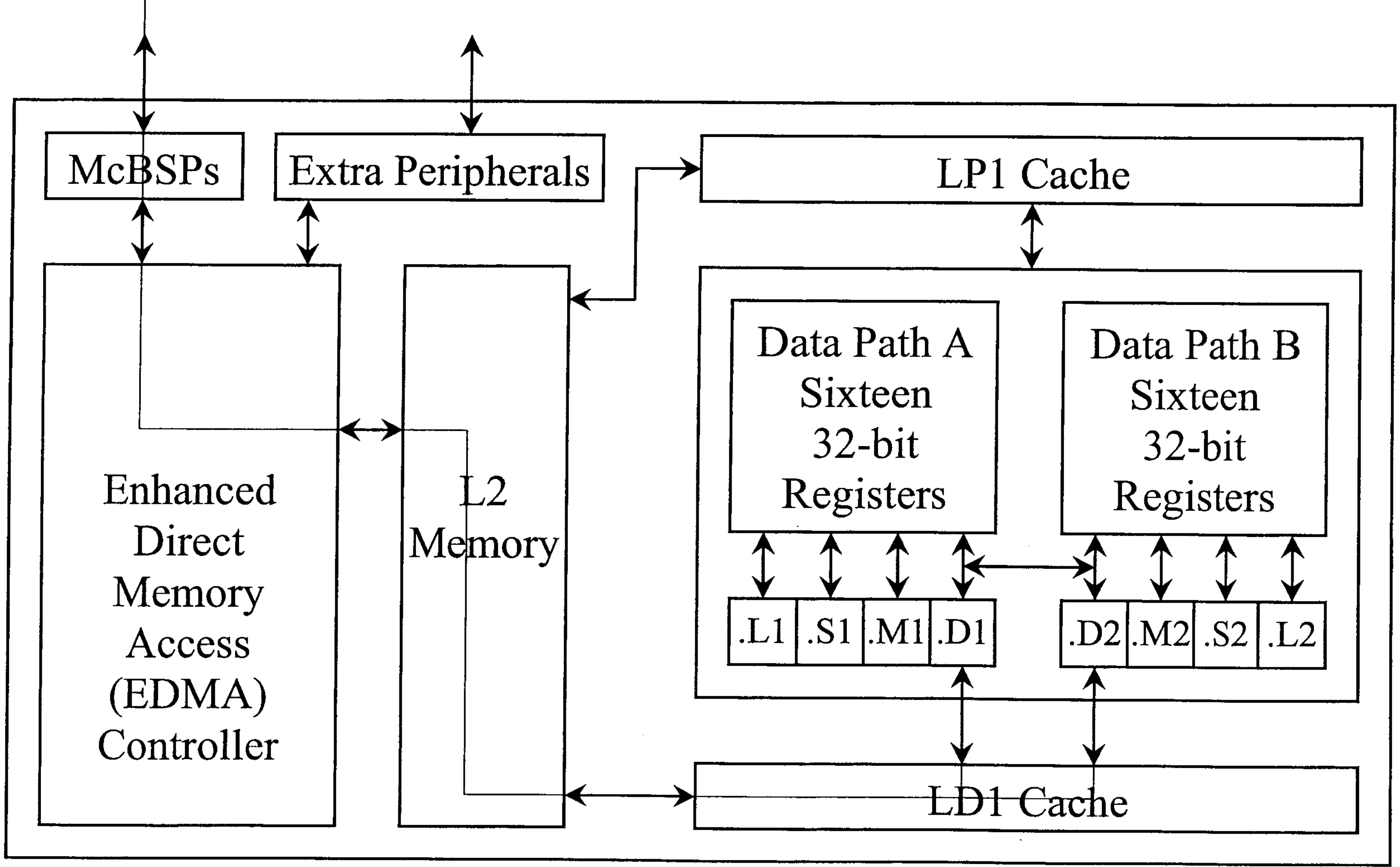
.S : fixed & floating-pt. ALU, branch instructions, and bit manipulations

.M : fixed & floating-pt. multiplier

.D : fixed-pt. only ALU and data transfers (loading/storing)

- Functional Units may only operate on values stored in an associated data path
- Two data cross paths (one in each direction).

TMS320C6711 DSP Chip



3.2 Mapping Between Instructions and Functional Units

Table 3–2 shows the mapping between instructions and functional units and Table 3–3 shows the mapping between functional units and instructions.

Table 3–2. Instruction to Functional Unit Mapping

.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 NRP†	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

4.2 Mapping Between Instructions and Functional Units

Table 4–2 shows the mapping between instructions and functional units and and Table 4–3 shows the mapping between functional units and instructions.

Table 4–2. Instruction to Functional Unit Mapping

.L Unit	.M Unit	.S Unit	.D Unit
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
DPTRUNC		CMPGTDP	
INTDP		CMPGTSP	
INTDPU		CMPLTDP	
INTSP		CMPLTSP	
INTSPU		RCPDP	
SPINT		RCPSP	
SPTRUNC		RSQRDP	
SUBDP		RSQRSP	
SUBSP		SPDP	

- At each clock cycle, each functional unit may be used on their associated data path.

- Also, at each cycle, one register in the "other" data path may be used provided the result is stored in the "original" data path

- 8 instructions may be executed during each clock cycle (due to the 8 functional units)

- 6 floating-point operations (FLOPs), or instructions, may be executed during each clock cycle

- The CPU is running @ rate 150 MHz which means that the C6711 may execute up to

  $8 \times 150 \text{ (MHz)} = 1200$ Million Instructions Per Second (MIPS)

  $6 \times 150 \text{ (MHz)} = 900$ MFLOPs

- Most arithmetic, logic, and branch operations require more than one clock cycle to complete (especially floating-point instructions).

do example

- ~~All assembly instructions may be conditional based on whether or not the value in a conditional register is zero. The conditional registers are B0, B1, B2, A1, A2.~~

-

A programming example

$$y = 1 + 2 + \dots + N = \sum_{K=1}^{N} K = \frac{N(N+1)}{2}$$

$$\begin{array}{rl} & y = 1 + 2 + \dots + (N-1) + N \\ + & y = N + (N-1) + \dots + 2 + 1 \\ \hline & 2y = \underbrace{(N+1) + (N+1) + \dots + (N+1) + (N+1)}_{N - (N+1) \text{ terms}} = N(N+1) \end{array}$$

$$2y = N(N+1)$$

$$y = \frac{N(N+1)}{2}$$

Go through programming example

- calculate the sum $\sum_{k=1}^{N} k = 1 + \ldots + N = \frac{N(N+1)}{2}$

- for $N = 3$, $\sum_{k=1}^{3} k = 1 + 2 + 3 = 6$ or $\frac{3(3+1)}{2} = \frac{12}{2} = 6$

```
1.)   // sum_C.c
2.)   // calculates a sum by calling a C function
3.)
4.)   #include <stdio.h>              // required for printing output
5.)   short sum_c_func(short k);      // prototype for C function
6.)
7.)   short N=3;                      // number of integers to sum
8.)   short sum;                      // store value returned from sumcfunc
9.)
10.)  short sum_c_func(short k)
11.)  {
12.)   short i;
13.)   short total = 0;
14.)
15.)   for (i=k; i>0; i--)            // sum from top to bottom
16.)   {
17.)    total += i;
18.)   }
19.)
20.)   return(total);
21.)  }
22.)
23.)  void main()
24.)  {
25.)    sum = sum_c_func(N);          // call sumcfunc to calculate sum
26.)    printf("Sum = %d", sum);      // print result
27.)  }
```

```
1.)   // sum_asm.c
2.)   // Calculates a sum by calling an assembly function
3.)
4.)   #include <stdio.h>
5.)   extern short sum_asm_func();   // declare external assembly function
6.)
7.)   short N=3;                     // number of integers to sum
8.)   short sum;                     // value returned from sum_asm_func
9.)
10.)  main()
11.)  {
12.)   sum = sum_asm_func(N);        // call sum_asm_func to calculate sum
13.)   printf("Sum = %d", sum);      // print result
14.)  }
```

```
1.)   ; sum_asm_func.asm
2.)   ; assembly function to find n+(n-1)+...+1
3.)
4.)                       .def           _sum_asm_func   ; asm function called from C
5.)   _sum_asm_func:      MV      .L1    A4,A1           ; setup N as loop counter in A1
6.)                       SUB     .S1    A1,1,A1         ; decrement N
7.)
8.)   LOOP:               ADD     .L1    A4,A1,A4        ; accumulate in A4
9.)                       SUB     .S1    A1,1,A1         ; decrement loop counter A1
10.)        [A1]          B       .S2    LOOP            ; branch to LOOP if A1#0
11.)                      NOP            5               ; five NOPs for delay slots
12.)                      B       .S2    B3              ; return to calling routine
13.)                      NOP            5               ; five NOPs for delay slots
14.)                      .end
```

Sum_asm (N=3)

registers allocated by calling function (B3, A4, B4, A6 ...); programmer allocated (A1)

	Clock Cycle(s)	B3	A4	B4	A6	A1	Instruction	Functional Unit	Operand
	0	return address (HEX	3	x	x	x	.def		_sum_asm_func
	1	↓	3	x	x	3	MV (move)	.L1 (.S1, .D1)	A4, A1
	2		3	x	x	2	SUB (subtract)	.S1 (.L1, .S1)	A1, 1, A1
LOOP	3		5	x	x	2	ADD	.L1 (.S1, .D1)	A4, A1, A4
	4		5	x	x	1	SUB	.S1 (.L1, .D1)	A1, 1, A1
	5		5	x	x	1	B (branch or jump)	.S2 (.S1)	LOOP (since A1 ≠ 0)
	6 - 10		5	x	x	1	NOP (no operation)		
LOOP	11		6	x	x	1	ADD	.L1 (.S1, .D1)	A4, A1, A4
	12		6	x	x	0	SUB	.S1 (.L1, .D1)	A1, 1, A1
	13		6	x	x	0	B	.S2 (.S1)	Ignore since A1 = 0
	14-18		6	x	x	0	NOP		
return	19		6	x	x	0	B	.S2 (X)	B3 (return from fcn)
	20-24	↓	(6)	x	x	0	NOP		

- Only one value may be returned from a function. By default, the value in A4 at the end of the 5th clock cycle of the return from function call (i.e. B .S2 B3) will be returned.

- The branch instruction used to return from a function call <u>must</u> use the functional unit .S2

```
1.)  ; sum_asm_func.asm
2.)  ; assembly function to find n+(n-1)+...+1
3.)
4.)                      .def         _sum_asm_func  ; asm function called from C
5.)  _sum_asm_func:      MV     .L1   A4,A1          ; setup N as loop counter in A1
6.)                      SUB    .S1   A1,1,A1        ; decrement N
7.)
8.)  LOOP:               ADD    .L1   A4,A1,A4       ; accumulate in A4
9.)                      SUB    .S1   A1,1,A1        ; decrement loop counter A1
10.)        [A1]         B      .S2   LOOP           ; branch to LOOP if A1#0
11.)                     NOP          5              ; five NOPs for delay slots
12.)                     B      .S2   B3             ; return to calling routine
13.)                     NOP          5              ; five NOPs for delay slots
14.)                     .end
```

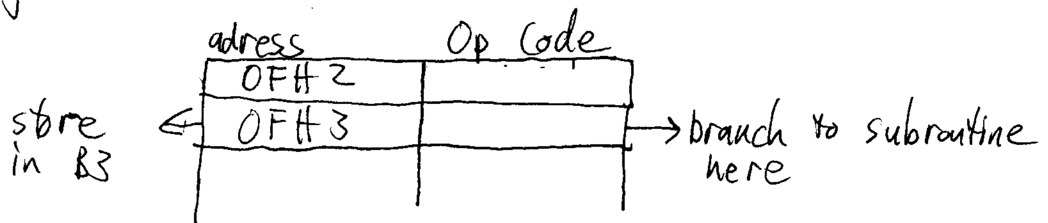
```
1.)  ; sum_asm_func.asm using cross-paths and alternative branch instruction
2.)  ; assembly function to find n+(n-1)+...+1
3.)
4.)                      .def         _sum_asm_func  ; asm function called from C
5.)  _sum_asm_func:      MV     .L2x  A4,B1          ; setup N as loop counter in A1
6.)                      SUB    .S2   B1,1,B1        ; decrement N
7.)
8.)  LOOP:               ADD    .L1x  A4,B1,A4       ; accumulate in A4
9.)                      SUB    .S2   B1,1,B1        ; decrement loop counter A1
10.)        [B1]         B      .S1   LOOP           ; alternative branch unit .S1
11.)                     NOP          5              ; five NOPs for delay slots
12.)                     B      .S2   B3             ; return to calling routine
13.)                     NOP          5              ; .S2 must be used here
14.)                     .end
```

```
1.)   ; sum_asm_func.asm
2.)   ; assembly function to find n+(n-1)+...+1
3.)
4.)                       .def           _sum_asm_func  ; asm function called from C
5.)   _sum_asm_func:      MV     .L1     A4,A1          ; setup N as loop counter in A1
6.)                       SUB    .S1     A1,1,A1        ; decrement N
7.)
8.)   LOOP:               ADD    .L1     A4,A1,A4       ; accumulate in A4
9.)                       SUB    .S1     A1,1,A1        ; decrement loop counter A1
10.)        [A1]          B      .S2     LOOP           ; branch to LOOP if A1#0
11.)                      NOP            5              ; five NOPs for delay slots
12.)                      B      .S2     B3             ; return to calling routine
13.)                      NOP            5              ; five NOPs for delay slots
14.)                      .end
```

```
1.)   ; sum_sa_func.sa
2.)   ; linear assembly function to find n+(n-1)+...+1
3.)
4.)                     .def        _sum_sa_func   ; linear asm function called from C
5.)   _sum_sa_func:     .cproc      N              ; start of linear asm function
6.)                     .reg        sum,ctr        ; asm optimizer directive
7.)                     MV          N,ctr          ; setup loop counter in ctr
8.)                     MV          N,sum          ; accumulate total in sum
9.)                     SUB         ctr,1,ctr      ; decrement ctr
10.)
11.)  LOOP:             ADD         sum,ctr,sum    ; accumulate in sum
12.)                    SUB         ctr,1,ctr      ; decrement loop counter
13.)        [ctr]       B           LOOP           ; branch to loop if ctr # 0
14.)                    .return     sum            ; return sum to calling function
15.)                    .endproc                   ; end of linear asm function
```

What happens when an assembly language function is called by another function?

1.) The current execution state (physical memory location in the program memory) is store in register B3.



2.) The values being passed to the function are stored in A4, B4, A6, ... as necessary for arguments that are 32-bits or less in length

(40-bit long int and 64-bit double-precision fp are stored in A4:A5, B4:B5, etc.)

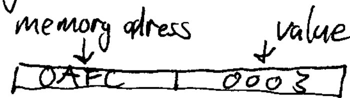
3.) The function processes the data stored in the registers (and possibly data loaded into the registers from memory) using ALU's and multipliers. The designation of a specific ALU or multiplier is done through assigning functional units to an instruction

4.) Upon completion, the function restores the execution state (stored in B3) and (possibley) returns the value stored in A4 to the calling function.

Notes

If more than one value needs to be ~~stored~~ returned (or stored) from a function call, the memory address of a variable may be passed to a function. The function may then store a processed value at the memory address of the variable. This is the equivalent of storing a value in that variable

For example, in data memory the value N = 3 may be stored.



subtract_one(&N)

↓

A4 = 0AFC

in asm

```
LDH  *A4, A6
NOP   4

SUB  A6, 1, A6
STH  *A4, A6
NOP

B  .S2  B3
NOP     5
```

N = subtract_one (N)

↓

A4 = 3

```
SUB  A4, 1, A4

B  .S2  B3
NOP     5
```

- <u>Conditional Registers</u>: All assembly language instructions may be conditional based on whether or not the value in a conditional register is equal to zero. The conditional registers are A1, A2, B0, B1, B2.

  [B0] $\leftrightarrow$ instruction is executed if $B0 \neq 0$, and

  [!B0] $\leftrightarrow$ " " " " $B0 = 0$,

  otherwise, the instruction is ignored.

- If two or more instructions need to executed and the output of one is not required for any of the others, they may be executed in parallel provided there are available functional units.

- <u>Data Cross Paths</u>: Data cross paths are used when the data that needs to be processed lies in both Data Paths A & B. The number in the functional unit of an instruction using cross-data paths (1 or 2) depends upon the Data Path where the data is being store (A or B respectively). Up to two data paths may be used at each clock cycle (one in each direction)

Assembly Language Column Format

Label || [ ] Instruction Functional Unit Operand ;

- || ← designate that this instruction is to execute in parallel with the previous instruction
- [ ] ← specify a conditional register
- Instruction ← fixed-or floating-point
- Operand ← registers being operated on
- ; ← comments

C vs. Assembly

C — High Level

- easiest to program
- hardware independent (compilers handle this)
- least efficient, however with better compilers, this is not as big of an issue as it once was

Linear Assembly (.sa)

- programs are broken down into individual operations
- allows for high-level "register" names
- pre-compiler optimizes register use (-o3 in build options

Assembly (.asm)

- programs are broken down into individual operations
- registers & and ALU's assigned to each operation
- programmer may hand optimize and perform pipeline scheduling

DSK features not used in this class

DMA - extendend memory
Flash Memory
DSP/BIOS
Software Scheduling / Pipelining