ECE4703-B06
TMS320C6713 Architecture
Overview and Assembly Language Programming

D. Richard Brown III
Associate Professor
Worcester Polytechnic Institute
Electrical and Computer Engineering Department
drb@ece.wpi.edu

16-November-2006
Outline

- Letting CCS optimize your code for you
- Not good enough? Writing your own assembly language functions for the C6x
- C6x architecture specifics:
  - Registers
  - Functional units
  - Pipelining
  - Fetch/execute packets
Profiling Your Code in CCS

**Hint:** edit the columns to see averages
What does it mean?

- **Access count** is the number of times that CCS profiled the function
  - Note that the function was probably called more than 49 times. CCS only timed it 49 times.

- **Inclusive average** is the average number of cycles needed to run the function *including* any calls to subroutines

- **Exclusive average** is the average number of cycles needed to run the function *excluding* any calls to subroutines
Optimizing Compiler
Profiling results after compiler optimization

- In this example, we get a 3x-4x improvement with “Speed Most Critical” and “File (-o3)” optimization
- Optimization gains can sometimes be much larger, e.g. 20x
Limitations of hardware profiling

- Breakpoint/clock profiling method may not work with compiler-optimized code
- Profile -> View method is known to be somewhat inaccurate when connected to real hardware (see “profiling limitations” in CCS help)
  - Accuracy is better when only one or two ranges are profiled
  - Best accuracy is achieved by running a simulator
Other factors affecting code efficiency

- **Memory**
  - C6713 has 64kB internal ram (L2 cache)
  - DSK provides additional 16MB external RAM (SDRAM)
  - Code location (.text in command file)
    - internal memory (fast)
    - external memory (slow, typically 2-4x worse)
  - Data location (.data in command file)
    - internal memory (fast)
    - external memory (slow, depends on datatypes)

- **Data types**
  - Slowest execution is double-precision floating point
  - Fastest execution is fixed point, e.g. short
Command file example

MEMORY
{
  vecs:  o = 00000000h  l = 00000200h
  IRAM: o = 00000200h  l = 0000FE00h
  CE0:  o = 80000000h  l = 01000000h
}

SECTIONS
{
  "vectors" > vecs
  .cinit > IRAM
  .text > IRAM
  .stack > IRAM
  .bss > IRAM
  .const > IRAM
  .data > IRAM
  .far > IRAM
  .switch > IRAM
  .sysmem > IRAM
  .tables > IRAM
  .cio > IRAM

  Code goes here
  Data goes here

  Addresses 00000000-0000FFFF are mapped to internal memory (IRAM). This is 64kB.
  External memory (CE0) is mapped to address range 80000000 – 80FFFFFF. This is 16MB.
### TMS320C6000 C/C++ Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Representation</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char</td>
<td>8 bits</td>
<td>ASCII</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8 bits</td>
<td>ASCII</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32768</td>
<td>32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>65535</td>
</tr>
<tr>
<td>int, signed int</td>
<td>32 bits</td>
<td>2s complement</td>
<td>-2147483648</td>
<td>214783647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32 bits</td>
<td>Binary</td>
<td>0</td>
<td>4294967295</td>
</tr>
<tr>
<td>long, signed long</td>
<td>40 bits</td>
<td>2s complement</td>
<td>-549755813888</td>
<td>549755813887</td>
</tr>
<tr>
<td>unsigned long</td>
<td>40 bits</td>
<td>Binary</td>
<td>0</td>
<td>1099511627775</td>
</tr>
<tr>
<td>enum</td>
<td>32 bits</td>
<td>2s complement</td>
<td>-2147483648</td>
<td>214783647</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175494e-38†</td>
<td>3.40282346e+38</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>IEEE 64-bit</td>
<td>2.22507385e-308†</td>
<td>1.79769313e+308</td>
</tr>
<tr>
<td>long double</td>
<td>64 bits</td>
<td>IEEE 32-bit</td>
<td>2.22507385e-308†</td>
<td>1.79769313e+308</td>
</tr>
</tbody>
</table>
Sometimes you have to take matters into your own hands...

Three options:

1. Linear assembly (.sa)
   - Compromise between effort and efficiency
   - Typically more efficient than C
   - Assembler takes care of details like assigning “functional units”, registers, and parallelizing instructions

2. ASM statement in C code (.c)
   - asm(“assembly code”)

3. C-callable assembly function (.asm)
   - Full control of assigning functional units, registers, parallelization, and pipeline optimization
C-Callable Assembly Language Functions

- **Basic concepts:**
  - Arguments are passed in via registers A4, B4, A6, B6, ... in that order.
  - Result returned in A4 also.
  - Return address of calling code (program counter) is in B3. Don’t overwrite B3!
  - Naming conventions:
    - In C code: label
    - In ASM code: _label (note the leading underbar)
  - Accessing global variables in ASM:
    - .ref _variablename
Skeleton C-Callable ASM Function

; header comments
; passed in parameters in 32-bit registers A4, B4, A6, ... in that order

.def _myfunc ; allow calls from external
ACONSTANT .equ 100 ; declare constants
.ref _aglobalvariable ; refer to a global variable

_myfunc: NOP ; instructions go here
B B3 ; return (branch to addr B3)
NOP 5 ; function output will be in A4
NOP 5 ; pipeline flush

.end
Example C-Callable Assembly Language Program

;charset file name

.void _fircasmfunc

A6,A1    ; set loop count
A6,2,A6   ; since dly buffer data as byte
A8       ; init A8 for accumulation
A6,B4,B4 ; since coeff buffer data as byte
B4,1,B4  ; B4 = bottom coeff array h[N-1]

LDH *A4++,A2 ; A2 = x[n-(N-1)+i] i=0,1,...,N-1
LDH *B4--,B2 ; B2 = h[N-1-i] i=0,1,...,N-1
NOP 4
MPY A2,B2,A6 ; A6 = x[n-(N-1)+i]*h[N-1-i]
NOP
ADD A6,A8,A8 ; accumulate in A8
LDH *A4,A7 ; A7 = x[(n-(N-1)+i+1] update delays
NOP 4 ; using data move "up"
STH A7,*-A4[1] ; --x[(n-(N-1)+i] update sample
SUB A1,1,A1 ; decrement loop count
[A1] B loop ; branch to loop if count # 0
NOP 5
MV A8,A4 ; result returned in A4
B B3 ; return addr to calling routine
NOP 4
Writing Efficient Assembly Language Programs for the C6x

- Need to become familiar with:
  - Specific architecture, capabilities, and limitations of the C6x
    - Registers
    - Functional units
    - Pipeline
    - Parallelization
    - ...
  - Instruction set
One instruction is 32 bits. Program bus is 256 bits wide.

Can execute up to 8 instructions per clock cycle (225MHz->4.4ns clock cycle).

8 independent functional units:
- 2 multipliers
- 6 ALUs

Code is efficient if all 8 functional units are always busy.

Register files each have 16 general purpose registers, each 32-bits wide (A0-A15, B0-B15).

Data paths are each 64 bits wide.
C6713 Functional Units

- Two data paths (A & B)
- **Data path A**
  - Multiply operations (\texttt{M1})
  - Logical and arithmetic operations (\texttt{L1})
  - Branch, bit manipulation, and arithmetic operations (\texttt{S1})
  - Loading/storing and arithmetic operations (\texttt{D1})
- **Data path B**
  - Multiply operations (\texttt{M2})
  - Logical and arithmetic operations (\texttt{L2})
  - Branch, bit manipulation, and arithmetic operations (\texttt{S2})
  - Loading/storing and arithmetic operations (\texttt{D2})
- All data (not program) transfers go through \texttt{D1} and \texttt{D2}
Fetch & Execute Packets

- C6713 fetches 8 instructions at a time (256 bits)
- **Definition:** “Fetch packet” is a group of 8 instructions fetched at once.
- Coincidentally, C6713 has 8 functional units.
  - Ideally, all 8 instructions would be executed in parallel.
- Often this isn’t possible, e.g.:
  - 3 multiplies (only two .M functional units)
  - Results of instruction 3 needed by instruction 4 (must wait for 3 to complete)
Execute Packets

- **Definition**: “Execute Packet” is a group of (8 or less) consecutive instructions in one fetch packet that can be executed in parallel.

- C compiler provides a flag to indicate which instructions should be run in parallel.

- You have to do this **manually** in Assembly using “||”.
C6713 Instruction Pipeline Overview

All instructions flow through the following steps:

1. **Fetch**
   a) PG: Program address Generate
   b) PS: Program address Send
   c) PW: Program address ready Wait
   d) PR: Program fetch packet Receive

2. **Decode**
   a) DP: Instruction DisPatch
   b) DC: Instruction DeCode

3. **Execute**
   a) 10 phases labeled E1-E10
   b) Fixed point processors have only 5 phases (E1-E5)

*Figure 7–5. Floating-Point Pipeline Phases*

`each step = 1 clock cycle`
Pipelining: Ideal Operation

Remarks:
- At clock cycle 11, the pipeline is “full”
- There are no holes (“bubbles”) in the pipeline in this example
Pipelining: “Actual” Operation

Remarks:
- Fetch packet $n$ has 3 execution packets
- All subsequent fetch packets have 1 execution packet
- Notice the holes/bubbles in the pipeline caused by lack of parallelization
Fetch Stage of C6713 Pipeline

Figure 7–2. Fetch Phases of the Pipeline

(a) PG | PS | PW | PR

PG: Program Address Generate
PS: Program Address Send
PW: Program Address Ready Wait
PR: Program Fetch Packet Receive
Decode Stage of C6713 Pipeline

- **DP** (instruction dispatch) phase
  - FPs are split into EPs
  - Instructions in an EP are assigned to appropriate functional units for decoding
- **DC** (instruction decode) phase: convert instruction to microcode for appropriate functional unit

*Figure 7–3. Decode Phases of the Pipeline*

(a) DP  DC

(b) Decode  32  32  32  32  32  32  32  32  32  32  32

ADD  ADD  STW  STW  ADDK  NOP†  DP

DC

MPYH


Functional units


† NOP is not dispatched to a functional unit.
Execute Stage of C6713 Pipeline

Figure 7-4. Execute Phases of the Pipeline and Functional Block Diagram of the TMS320C67x

(a) E1 E2 E3 E4 E5 E6 E7 E8 E9 E10

(b) [Diagram of Execute stage with block diagrams and labels for SADD, B, SMPY, STH, etc.]

WPI
Execute Stage of C6713 Pipeline

- C67x has 10 execute phases (floating point)
  
  ![Floating-Point Pipeline Phases](image)

- C62x/C64x have 5 execute phases (fixed point)
- Different types of instructions require different numbers of these phases to complete their execution
  - Anywhere between 1 and all 10 phases
  - Most instruction tie up their functional unit for only one phase (E1)
Execute Stage: Delay Slots

- How long must we wait for the result of an instruction?
  - Most instructions’ results are available at the end of E1 (called “single-cycle” instructions)
    - Examples:
      - ABSSP (single precision absolute value)
      - RCPSP (single precision reciprocal approximation)
  - Some instructions take more time to produce results
    - Examples:
      - MPYSP (single precision multiply): Results available at the end of E4 (3 delays slots)
      - ADDSP (single precision addition): Results available at the end of E4 (3 delay slots)
Execute Stage: Functional Latency

- How long must we wait for the functional unit to be free?
  - Most instructions tie up the functional unit for only one pipeline stage (E1)
    - Examples:
      - All single-cycle instructions
      - Most multicycle instructions, including, for example, ADDSP (single precision addition)
  - Some instructions tie up the execution unit for more than one pipeline stage
    - Examples:
      - MPYDP (double precision multiply): .M execution unit is tied up for 4 pipeline stages (E1-E4). Can’t use this functional unit until E4 completes.
## Execution Stage Examples (1)

### ABSSP

**Single-Precision Floating-Point Absolute Value**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>.unit (src2, dst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>.unit</td>
<td>.S1 or .S2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode map field used...</th>
<th>For operand type...</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>src2</td>
<td>xsp</td>
<td>.S1, .S2</td>
</tr>
<tr>
<td>dst</td>
<td>sp</td>
<td></td>
</tr>
</tbody>
</table>

### Pipeline

- **Pipeline Stage**: E1
- **Read**: src2
- **Written**: dst
- **Unit in use**: .S

**results available after E1 (zero delay slots)**

**Functional unit free after E1**

(1 functional unit latency)
**Execution Stage Examples (2)**

**ADDSP**

**Single-Precision Floating-Point Addition**

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Read</strong></td>
<td>src1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>src2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Written</strong></td>
<td></td>
<td></td>
<td>dst</td>
<td></td>
</tr>
<tr>
<td><strong>Unit in use</strong></td>
<td></td>
<td></td>
<td>.L</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax**

ADDSP (.unit) src1, src2, dst

.unit = .L1 or .L2

**Instruction Type**

4-cycle

**Delay Slots**

3

**Functional Unit Latency**

1

**Results**

- Functional unit free after E1 (1 functional unit latency)
- Results available after E4 (3 delay slots)
## Execution Stage Examples (3)

### Single-Precision Floating-Point Multiply

**Syntax**

MPYSP (.unit) src1, src2, dst

.unit = .M1 or .M2

**Pipeline**

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>src1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>src2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Written</td>
<td></td>
<td></td>
<td></td>
<td>dst</td>
</tr>
<tr>
<td>Unit in use</td>
<td>.M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If dst is used as the source for the ADDDP, CMPEQDP, CMPLTDP, CMPGTDP, MPYDP, or SUBDP instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type**

4-cycle

**Delay Slots**

3

**Functional Unit Latency**

1

Results available after E4 (3 delay slots)

Functional unit free after E1 (1 functional unit latency)
Execution Stage Examples (4)

Double-Precision Floating-Point Multiply

**MPYDP**

**Syntax**

MPYDP (.unit) src1, src2, dst

.unit = .M1 or .M2

**Pipeline**

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Stage</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>E4</th>
<th>E5</th>
<th>E6</th>
<th>E7</th>
<th>E8</th>
<th>E9</th>
<th>E10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>src1_i</td>
<td>src1_i</td>
<td>src1_h</td>
<td>src1_h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>src2_i</td>
<td>src2_h</td>
<td>src2_i</td>
<td>src2_h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Written</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>dst_i</td>
<td>dst_h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If *dst* is used as the source for the ADDDP, CMPEQDP, CMPLTDP, CMPGTDP, MPYDP, or SUBDP instruction, the number of delay slots can be reduced by one, because these instructions read the lower word of the DP source one cycle before the upper word of the DP source.

**Instruction Type**

MPYDP

**Delay Slots**

9

**Functional Unit Latency**

4

Results available after E10 (9 delay slots)

Functional unit free after E4 (4 functional unit latency)
Delay Slots & Functional Latency

- IMPORTANT: **Delay slots** are not the same as **functional unit latency**

- Example:

  ```
  MPYSP .M1 A1, A2, A3 ; A3 = A1 x A2
  MPYSP .M1 A4, A5, A6 ; A6 = A4 x A5
  MPYSP .M1 A7, A8, A9 ; A9 = A6 x A7
  MPYSP .M1 A10, A11, A12 ; A12 = A10 x A11
  ```

- Is this code ok?
Delay Slots & Functional Latency

- What about this code?

\[
\text{MPYSP .M1 A1, A2, A3} \quad ; \quad A3 = A1 \times A2 \\
\text{MPYSP .M1 A3, A4, A5} \quad ; \quad A5 = A3 \times A4
\]
Delay Slots & Functional Latency

- Won’t work because the result in A3 is not available until E4 completes for the first MPYSP instruction
- “Data hazard” due to the **delay slots** in MPYSP
- How to “fix” the last example

```plaintext
MPYSP .M1 A1, A2, A3 ; A3 = A1 x A2
NOP 3 ; insert 3 delay slots
        ; now ok to use A3
MPYSP .M1 A3, A4, A5 ; A5 = A3 x A4
```
Delay Slots & Functional Latency

- What about this code?

```
MPYDP .M1 A1:A0, A3:A2, A5:A4
MPYDP .M1 A7:A6, A9:A8, A11:A10
```
Delay Slots & Functional Latency

- Won’t work because the functional unit M1 is tied up for 4 clock cycles (E1-E4) by MPYDP
- “Resource conflict” due to the **functional latency** in MPYDP
- How to fix it:
  
  MPYDP .M1 A1:A0, A3:A2, A5:A4
  NOP 3 ; 3 NOPs for func latency
  MPYDP .M1 A7:A6, A9:A8, A11:A10
Delay Slots & Functional Latency

- What about this code?

```assembly
MPYDP .M1 A1:A0, A3:A2, A5:A4
MPYDP .M1 A5:A4, A8:A7, A11:A10
```
Delay Slots & Functional Latency

- Two problems now!
  - Resource conflict for .M1 unit (E2-E4)
  - Data hazard for result in A5:A4 (E2-E10)
- The “fix”:

```
MPYDP .M1 A1:A0, A3:A2, A5:A4
NOP         9
MPYDP .M1 A5:A4, A8:A7, A11:A10
```

- Note: Could use M1 after E4, but A5:A4 not available until after E10.
Functional Latency & Delay Slots

- **Functional Latency**: How long must we wait for the functional unit to be free?
- **Delay Slots**: How long must we wait for the result?
- **General remarks:**
  - Functional unit latency $\leq$ Delay slots
  - Strange results will occur in ASM code if you don’t pay attention to delay slots and functional unit latency
  - All problems can be resolved by “waiting” with NOPs
  - Efficient ASM code tries to keep functional units busy all of the time.
  - Efficient code is hard to write (and follow).
Additional Constraints: Data Cross-Paths

- TMS320C6x core has A side and B side
  - A side: M1, S1, L1, D1, and register file A0-A15
  - B side: M2, S2, L2, D2, and register file B0-B15

- Cross path instruction examples:
  - MPY .M1x A2, B2, A4 ; cross path brings B2 to M1
  - MPY .M2x A2, B2, B4 ; cross path brings A2 to M2

- Constraint: **Only two cross-paths are available per cycle:** 1→2 and 2→1.
  - Note: Can’t have two 1→2 or two 2→1 cross paths in the same cycle.
Additional Constraints

- **Memory constraints**
  - Two memory accesses can be performed in one cycle if they don’t access the same bank of memory
  - See textbook Section 3.20.1

- **Load/Store constraints**
  - Address register must agree with .D unit
    - e.g.: `LDW .D1 *A1, A2 ; valid because A1 and D1 agree`
  - Parallel loads and stores must use different register files
  - See textbook Section 3.20.3
Suggested Reading

- Reference material (on course web page)
  - TMS320C6000 CPU Instruction Set and Reference Guide
  - TMS320C6000 Programmer’s Guide
- Kehtarnavaz Chap 3
- Kehtarnavaz Chap 7
- Chassaing has some ASM example code in the myproject directory (see, for example, FIRcASM.pjt)