Project 5 Objectives

— Learning the architecture of a DSP
— Developing C code for the TMS320C6201
— Evaluating DSP performance
  * Processor architecture
  * C code optimizations
— Buffered serial interface
  * Flexible serial transfer
  * Contrast with 16F877
— Learn about Host-Port interface
— Examine C compilation for the PIC
Overview of DSP Architectures

— Useful for image, speech processing
— Signal processing applications require significant bandwidth, multiplications
— Most devices use Harvard architecture
— ALUs structured to perform multiply/accumulate
— Special instructions often allocated to DSPs
Harvard Architecture Example

DSP56000 Instruction/Data Path

- Separate instruction and data buses
- Multiple functional units
  * arithmetic operations performed in parallel
  * Very Long Instruction Word (VLIW) - multiple instructions in one word.
  * difficult for compiler to determine execution
- Multiple simultaneous data accesses.
Multiply-Accumulate Hardware

\[ a_0x(n) + a_1x(n - 1) + a_2x(n - 2) + ... + a_{N-1}x(n - (N - 1)) \]  \hspace{1cm} (1)

— *a* values are *coefficients*

— *x* values are data points

— Special path to adder for partial sums

— Useful for filtering applications
# DSP Applications

## The TMS320 Family of Digital Signal Processors

### Table 1–1. Typical Applications for the TMS320 Family of Digital Signal Processors (DSPs)

<table>
<thead>
<tr>
<th>Automotive</th>
<th>Consumer</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive ride control</td>
<td>Digital radios/TVs</td>
<td>Disk drive control</td>
</tr>
<tr>
<td>Antiskid brakes</td>
<td>Educational toys</td>
<td>Engine control</td>
</tr>
<tr>
<td>Cellular telephones</td>
<td>Music synthesizers</td>
<td>Laser printer control</td>
</tr>
<tr>
<td>Digital radios</td>
<td>Pagers</td>
<td>Motor control</td>
</tr>
<tr>
<td>Engine control</td>
<td>Power tools</td>
<td>Robotics control</td>
</tr>
<tr>
<td>Global positioning</td>
<td>Radar detectors</td>
<td>Servo control</td>
</tr>
<tr>
<td>Navigation</td>
<td>Solid-state answering machines</td>
<td></td>
</tr>
<tr>
<td>Vibration analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voice commands</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General-Purpose</th>
<th>Graphics/Imaging</th>
<th>Industrial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive filtering</td>
<td>3-D computing</td>
<td>Numeric control</td>
</tr>
<tr>
<td>Convolution</td>
<td>Animation/digital maps</td>
<td>Power-line monitoring</td>
</tr>
<tr>
<td>Correlation</td>
<td>Homomorphic processing</td>
<td>Robotics</td>
</tr>
<tr>
<td>Digital filtering</td>
<td>Image compression/transmission</td>
<td>Security access</td>
</tr>
<tr>
<td>Fast Fourier transforms</td>
<td>Image enhancement</td>
<td></td>
</tr>
<tr>
<td>Hilbert transforms</td>
<td>Pattern recognition</td>
<td></td>
</tr>
<tr>
<td>Waveform generation</td>
<td>Robot vision</td>
<td></td>
</tr>
<tr>
<td>Windowing</td>
<td>Workstations</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instrumentation</th>
<th>Medical</th>
<th>Military</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital filtering</td>
<td>Diagnostic equipment</td>
<td>Image processing</td>
</tr>
<tr>
<td>Function generation</td>
<td>Fetal monitoring</td>
<td>Missile guidance</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>Hearing aids</td>
<td>Navigation</td>
</tr>
<tr>
<td>Phase-locked loops</td>
<td>Patient monitoring</td>
<td>Radar processing</td>
</tr>
<tr>
<td>Seismic processing</td>
<td>Prosthetics</td>
<td>Radio frequency modems</td>
</tr>
<tr>
<td>Spectrum analysis</td>
<td>Ultrasound equipment</td>
<td>Secure communications</td>
</tr>
<tr>
<td>Transient analysis</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Telecommunications</th>
<th>Voice/Speech</th>
</tr>
</thead>
<tbody>
<tr>
<td>1200- to 56 600-bps modems</td>
<td>Faxing</td>
</tr>
<tr>
<td>Adaptive equalizers</td>
<td>Future terminals</td>
</tr>
<tr>
<td>ADPCM transcoders</td>
<td>Line repeaters</td>
</tr>
<tr>
<td>Base stations</td>
<td>Personal communications</td>
</tr>
<tr>
<td>Cellular telephones</td>
<td>systems (PCS)</td>
</tr>
<tr>
<td>Channel multiplexing</td>
<td>Personal digital assistants (PDA)</td>
</tr>
<tr>
<td>Data encryption</td>
<td>Speaker phones</td>
</tr>
<tr>
<td>Digital PBXs</td>
<td>Spread spectrum communications</td>
</tr>
<tr>
<td>Digital speech interpolation (DSI)</td>
<td>Digital subscriber loop (xDSL)</td>
</tr>
<tr>
<td>DTMF encoding/decoding</td>
<td>Video conferencing</td>
</tr>
<tr>
<td>Echo cancellation</td>
<td>X.25 packet switching</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1-4
2.1 TMS320C62x/C67x Block Diagram

The 'C62x/C67x processor consists of three main parts: CPU (or the core), peripherals, and memory. Eight functional units operate in parallel, with two similar sets of the basic four functional units. The units communicate using a cross path between two register files, each of which contains 16 32-bit registers. Program parallelism is defined at compile time because there is no data dependency checking done in hardware during run time. The 256-bit-wide program memory fetches eight 32-bit instructions every single cycle.

Figure 2–1 is the block diagram for the TMS320C62x/C67x devices. The 'C62x/C67x devices come with on-chip program and data memory, which may be configured as cache on some devices. Peripherals include a direct memory access (DMA) controller, power-down logic, external memory interface (EMIF), serial port(s), expansion bus or host port, and timer(s). Check the data sheet for your device to determine the specific peripheral configurations you have.

Figure 2–1. TMS320C62x/C67x Block Diagram
TMS320C6201 Highlights

— Operates at 200MHz clock speed
— 32 general purpose registers of 32 bits
— Architecture contains two multipliers and six ALUs
— Executes up to eight operations per cycle
  * eight 32 bit instructions
  * separate instruction/data bus
— RISC-like code to optimize control circuitry
DSP Data and Instruction Paths

— Four data buses of 32 bits
  * two buses dedicated to SRAM data fetch
  * two buses dedicated to SRAM data write

— 256 bit bus transports 8 instructions per clock cycle from instruction memory

— Key is finding 8 non-dependent instructions to be performed simultaneously

— 256 bit value referred to as VLIW since it controls eight functional units

— Eight grouped instructions referred to as an execute packet
Figure 2–2. TMS320C62x CPU Data Paths
TMS320C6201 Data Path

— Eight functional units broken into groups of 4
  * \( .L \) and \( .S \) units perform integer arithmetic
  * \( .M \) units perform floating point operations
  * \( .D \) units perform data transfer operations

— All units complete operation in one instruction cycle
2.2 Introduction to the Example Code

The C code example that you will use to start this tutorial is demo1.c, which is shown in Example 2–1. This example calls three functions: mac1(), vec_mpy1(), and iir1().

Example 2–1. The Code Example—demo1.c

```c
main(int argc, char *argv[])
{
    const short coefs[150];
    short optr[150];
    short state[2];
    const short a[150];
    const short b[150];
    int c = 0;
    int dotp[1] = {0};
    int sum= 0;
    short y[150];
    short scalar = 3345;
    const short x[150];

    sum = mac1(a, b, c, dotp);
    vec_mpy1(y, x, scalar);
    iir1(coefs, x, optr, state);
}
```

The mac1() function, a multiply accumulate and squaring accumulate example, is shown in Example 2–2. It is performing a dot product of vector a with vector b and is also squaring and summing vector b.

Example 2–2. The Multiply Accumulate Function—mac1.c

```c
int mac1(const short *a, const short *b, int sqr, int *sum)
{
    int i;
    int dotp = *sum;

    for (i = 0; i < 150; i++)
    {
        dotp += b[i] * a[i];
        sqr += b[i] * b[i];
    }

    *sum = dotp;
    return sqr;
}
```
Lesson 3: Phase 1 of the Code Development Flow

2.5 Lesson 3: Phase 1 of the Code Development Flow

Looking at the functions in demo1 one at a time, you can determine whether or not they need to be improved and, if they do need to be improved, how they can be improved. Start by looking at the first function, mac1().

Example 2–6 shows the assembly output of the function’s inner loop kernel. The loop kernel is the area of the loop with the most parallelism. Only the inner loop is shown, because this is the area that can be improved with software pipelining. Notice that there are eight instructions executing in parallel (as indicated by the seven sets of parallel bars). This is the maximum number of instructions that the ’C6x can execute in parallel, so this code does not need to be improved.

Example 2–6. Inner Loop Kernel of mac1.asm

```
L3:        ; PIPED LOOP KERNEL
    ADD     .L2     B4,B7,B7     ;
    ADD     .L1     A5,A3,A3     ;
    MPY     .M2X    A4,B5,B4     ;@@
    MPY     .M1     A4,A4,A5     ;@@
    [ B0]   B       .S1     L3           ;@@@@@
    [ B0]   SUB     .S2     B0,1,B0      ;@@@@@@
    LDH     .D1     *A0++,A4     ;@@@@@@@@
    LDH     .D2     *B6++,B5     ;@@@@@@@@
```

The @ characters specify the iteration of the loop that an instruction is on in the software pipeline; these symbols are automatically created by the code generation tools. The first iteration does not have an @ character; one @ character represents the second iteration; two @ characters represents the third iteration, and so on.

Because the mac1() function does not need to be improved, it does not need to go beyond phase 1 of the code development flow.
C Code Intrinsics

— Special compiler functions placed in C code
— Can be used to better identify parallelism
— Forces compiler to generate specific assembler instructions
— Examples when applied to 32 bit values:
  * _mpy(x1, x2) - multiply lower 16 bits of x1 with 16 LS bits of x2
  * _mpylh(x1, x2) - multiply lower 16 bits of x1 with 16 MS bits of x2
— Intrinsics start with leading _
2.6 Lesson 4: Phase 2 of the Code Development Flow

For your convenience, the vec_mpy1() function is duplicated here as Example 2–9 (the C version) and Example 2–10 (the assembly output of the inner loop). This is the same code that you saw in Example 2–3 and Example 2–7.

Example 2–9. The Vector Multiply Function—vec_mpy1.c

```c
void vec_mpy1(short y[], const short x[], short scalar)
{
    int i;
    for (i = 0; i < 150; i++)
        y[i] += ((scalar * x[i]) >> 15);
}
```

Example 2–9 uses short data types. Because short data types are 16 bits, they translate into halfword instructions, such as LDH and STH (see Example 2–10).

The loop in Example 2–10 uses two LDH instructions and an STH instruction to load x[i] and y[i] and store back to y[i]. Because only two memory operations can occur per cycle, the fastest that this loop can execute is one y[i] result every two cycles. The performance of this loop is limited by the number of D units.

Example 2–10. Inner Loop Kernel of vec_mpy1.asm

```
L3:        ; PIPED LOOP KERNEL
        ADD .L2X A3, B6, B5 ;
        |   B .S1 L3 ;
        |   LDH .D2 *B4(6), B6 ;@@
        |   LDH .D1 *A0++, A4 ;@@@@
        |   SHR .S1 A3, 15, A3 ;@
        |   MPY .M1 A5, A4, A3 ;@@
        |   SUB .L1 A1, i, A1 ;@@
```

Because x is an array, x[i] and x[i + 1] are next to each other in memory. This means that instead of using halfword instructions (LDH and STH) to load and store each element in the array, you can use word instructions (LDW and STW) to load and store two elements at a time, as long as the data is aligned on a word boundary. In other words, all word accesses should have the 2 LSBs of the address set to 0. Two elements at a time, x[i] and x[i + 1], fit into one 32-bit register.
Vector Multiply Example

Inner loop requires:

— 1 16 bit short $x$ value fetched
— 1 16 bit operations performed
— 1 short value written to memory

Idea:

— Read consecutive locations $x[i]$, $x[i+1]$ as a 32 bit integer
— Perform 2 16 bit operations
— Write both $y[i]$, $y[i+1]$ back to memory as a word
Vector Multiply with Intrinsics

Lesson 4: Phase 2 of the Code Development Flow

To achieve this in C, declare x[i] as an integer instead of as a short data type. Also, you need to use some intrinsics.

Now that you have determined that you can load x[i] and x[i + 1] into the same register, you need to figure out how to do it. You can do this by using the _mpy and _mpylh intrinsics. Intrinsics are like built-in C functions that correspond to 'C6x assembly language instructions. The _mpy intrinsic multiplies the 16 LSBs of one operand by the 16 LSBs of another and returns the result. The _mpylh intrinsic multiplies the 16 LSBs of the first operand by the 16 MSBs of the second and returns the result.

You can then use the _add2 intrinsic to add the 16 MSBs of the first operand to the 16 MSBs of the second operand. At the same time, the _add2 intrinsic also adds the 16 LSBs of the first operand to the 16 LSBs of the second operand. The result of both additions is stored in a 32-bit operand.

Example 2–11 shows how to rewrite the vec_mpy() function to include the _mpy and _mpylh intrinsics:

Example 2–11. The Revised Vector Multiply Function—vec_mpy2.c

```c
void vec_mpy2(int y[], const int x[], short scalar)
{
  int i, val;
  unsigned int temph, templ;
  for (i = 0; i < 75; i++)
  {
    val = x[i];
    templ = (_mpy  (scalar, val) >> 15) & 0x0000ffff;
    temph = (_mpylh(scalar, val) << 1 ) & 0xffff0000;
    y[i] = _add2(y[i], temph | templ);
  }
}
```
Vector Multiply Assembly Code

Lesson 4: Phase 2 of the Code Development Flow

The inner loop of the vec_mpy2( ) function translates into the assembly output shown in Example 2–15.

Example 2–15. Inner Loop Kernel of vec_mpy2.asm

```
L3:        ; PIPED LOOP KERNEL
    OR      .L2X    B5,A8,B7     ;@  
    ||         SHL     .S1     A6,1,A4      ;@@  
    || 
    ||         AND     .L1     A5,A4,A6     ;@@  
    ||         LDW     .D2     *(A4(12)),B5  ;@@@@
    ||         MPYLH   .M1     A0,A9,A6     ;@@@@
    ||         LDW     .D1     *(A3++),A9     ;@@@@@@
    ||         STW     .D2     B6,*B4++     ;
    ||         ADD2    .S2     B5,B7,B6     ;@  
    ||         AND     .L1     A7,A4,A8     ;@@
    ||         MV      .L2X    A6,B5        ;@@  
    ||         SUB     .D1     A1,1,A1      ;@@@  
    ||         SHR     .S1     A8,15,A8     ;@@@  
    ||         MPY      .M1     A0,A9,A8     ;@@@@
```

As you can see, the code for the vec_mpy2( ) function is improved over the original vec_mpy( ) code. Two LDW instructions are loading four elements (x[i], x[i+1], y[i], and y[i+1]), and one STW instruction is storing two elements: x[i] and y[i+1]. With the revised code, two y[i] results are stored every two cycles. Recall that only one y[i] result was stored every two cycles in Example 2–10.

Table 2–3 shows how the vec_mpy( ) function has improved as it moves from phase 1 to phase 2.

Table 2–3. Revised Cycle Counts for vec_mpy( )

<table>
<thead>
<tr>
<th>Function</th>
<th>Execute Packets</th>
<th>Loop Iterations</th>
<th>Constant</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec_mpy1( )</td>
<td>2</td>
<td>150</td>
<td>16</td>
<td>2 × 150 + 16 = 316</td>
</tr>
<tr>
<td>vec_mpy2( )</td>
<td>2</td>
<td>75</td>
<td>22</td>
<td>2 × 75 + 22 = 172</td>
</tr>
</tbody>
</table>
11.2 McBSP Interface Signals and Registers

The multichannel buffered serial port (McBSP) consists of a data path and a control path, which connect to external devices. Data is communicated to these external devices via separate pins for transmission and reception. Control information (clocking and frame synchronization) is communicated via four other pins. The device communicates to the McBSP via 32-bit-wide control registers accessible via the internal peripheral bus.

The McBSP consists of a data path and control path, as shown in Figure 11–1. Seven pins listed in Table 11–1 connect the control and data paths to external devices.

*Figure 11–1. McBSP Block Diagram*
Buffered Serial Port

— Transmit and receive data buffered
— Variable length data transfer
  (8, 12, 16, 20, 24, or 32) bits
— Independent data clocking
— Direct memory access
— Internal bit rate generator

Host-Port Interface
Figure 7–3 is a simplified diagram of the ‘C6201/C6701 HPI.

The HPI provides 32-bit data to the CPU with an economical 16-bit external interface by automatically combining successive 16-bit transfers. When the host device transfers data through HPID, the DMA auxiliary channel accesses the CPU’s address space.

The 16-bit data bus, HD[15:0], exchanges information with the host. Because of the 32-bit-word structure of the chip architecture, all transfers with a host consist of two consecutive 16-bit halfwords. On HPI data (HPID) write accesses, the HBE[1:0] byte enables select the bytes to be written. For HPIA, HPIC, and HPID read accesses, the byte enables are not used. The dedicated HHWIL pin indicates whether the first or second halfword is being transferred. An internal control register bit determines whether the first or second halfword is placed into the most significant halfword of a word. For a full word access, the host must not break the first halfword/second halfword (HHWIL low/high) sequence of an ongoing HPI access.

Figure 7–3. HPI Block Diagram
Host-Port Interface

— 16 bit external interface to 32 bit internal data bus
— Allows interface to external bus masters
— Permits direct memory access (DMA) for an external device
— Interrupt available to indicate when data is ready
C Compilation for the PIC 16F877

— PIC coding this semester in assembly language.
— Primitive shareware compiler available from B. Knudsen (www.bknd.com)
— Converts C code into assembly.
— Uses a DOS interface.
— Supported C structures are limited.
PIC Compilation Experiment

— We provide a simple *insertion sort* example.
— Student compiles, assembles and simulates example.
— Student measures number of required instruction cycles. via MPLAB simulation
— Student writes own assembly code to perform same insertion sort.
— Student assembly code is simulated with MPASM.
— Performance of the two approaches are compared.
Lab 5 Summary

— DSPs play an important role in many data-intensive systems
— Difficult to program high-level language to utilize DSP functional resources
— *Intrinsics* can be used for greater efficiency
— Advanced serial interface allows buffering of data
— Host interface allows simplified data transfer
— PIC C compiler gives a glimpse of how engineers target C to microcontrollers.