Assembly Languages

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science
Assembly Languages

One step up from machine language

Originally a more user-friendly way to program

Now mostly a compiler target

Model of computation: stored program computer
Assembly Language Model

PC → ALU ← Registers ← Memory

```
add r1, r2
sub r2, r3
cmp r3, r4
bne l1
sub r4, 1
l1: jmp l3
```
Assembly Language Instructions

Built from two pieces:

```
add R1, R3, 3
```

- **Opcode**: What to do with the data
- **Operands**: Where to get the data
Types of Opcodes

Arithmetic, logical
- add, sub, mult
- and, or
- Cmp

Memory load/store
- ld, st

Control transfer
- jmp
- bne

Complex
- movs
Operands

Each operand taken from a particular addressing mode:

Examples:

- Register: `add r1, r2, r3`
- Immediate: `add r1, r2, 10`
- Indirect: `mov r1, (r2)`
- Offset: `mov r1, 10(r3)`
- PC Relative: `beq 100`

Reflect processor data pathways
Types of Assembly Languages

Assembly language closely tied to processor architecture

At least four main types:

CISC: Complex Instruction-Set Computer
RISC: Reduced Instruction-Set Computer
DSP: Digital Signal Processor
VLIW: Very Long Instruction Word
CISC Assembly Language

Developed when people wrote assembly language

Complicated, often specialized instructions with many effects

Examples from x86 architecture

- String move
- Procedure enter, leave

Many, complicated addressing modes

So complicated, often executed by a little program (microcode)

Examples: Intel x86, 68000, PDP-11
RISC Assembly Language

Response to growing use of compilers
Easier-to-target, uniform instruction sets
“Make the most common operations as fast as possible”

Load-store architecture:

- Arithmetic only performed on registers
- Memory load/store instructions for memory-register transfers

Designed to be pipelined

Examples: SPARC, MIPS, HP-PA, PowerPC
DSP Assembly Language

Digital signal processors designed specifically for signal processing algorithms

Lots of regular arithmetic on vectors

Often written by hand

Irregular architectures to save power, area

Substantial instruction-level parallelism

Examples: TI 320, Motorola 56000, Analog Devices
VLIW Assembly Language

Response to growing desire for instruction-level parallelism
Using more transistors cheaper than running them faster
Many parallel ALUs
Objective: keep them all busy all the time
Heavily pipelined
More regular instruction set
Very difficult to program by hand
Looks like parallel RISC instructions
Examples: Itanium, TI 320C6000
Example: Euclid’s Algorithm

```c
int gcd(int m, int n)
{
    int r;
    while ((r = m % n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
```
### i386 Programmer’s Model

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
<td></td>
<td>cs</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
<td></td>
<td>ds</td>
</tr>
<tr>
<td></td>
<td>ecx</td>
<td></td>
<td>ss</td>
</tr>
<tr>
<td></td>
<td>edx</td>
<td></td>
<td>es</td>
</tr>
<tr>
<td></td>
<td>esi</td>
<td></td>
<td>fs</td>
</tr>
<tr>
<td></td>
<td>edi</td>
<td></td>
<td>gs</td>
</tr>
<tr>
<td></td>
<td>ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>esp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>eflags</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>eip</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **eax**: Mostly General-Purpose Registers
- **ebx**: Source index
- **ecx**: Destination index
- **edx**: Base pointer
- **esi** and **edi**: Stack pointer
- **ebp** and **esp**: Status word
- **eip**: Instruction Pointer
Euclid on the i386

.file "euclid.c"
.version "01.01"
gcc2_compiled.:
.text
.align 4
.globl gcd
.type gcd,@function
gcd:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%eax
movl 12(%ebp),%ecx
jmp .L6
.p2align 4,,7
Euclid on the i386

```
.file "euclid.c"
.version "01.01"
gcc2_compiled.:
.text
.align 4
.globl gcd
.type gcd,@function
gcd:
pushl %ebp
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%eax
movl 12(%ebp),%ecx
jmp .L6
.p2align 4,,7
```

Stack Before Call

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>8(%esp)</td>
</tr>
<tr>
<td>m</td>
<td>4(%esp)</td>
</tr>
<tr>
<td>R. A.</td>
<td>0(%esp)</td>
</tr>
</tbody>
</table>

Stack After Entry

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>12(%ebp)</td>
</tr>
<tr>
<td>m</td>
<td>8(%ebp)</td>
</tr>
<tr>
<td>R. A.</td>
<td>4(%ebp)</td>
</tr>
<tr>
<td>%ebp</td>
<td>old ebp</td>
</tr>
<tr>
<td>%esp</td>
<td>old ebx</td>
</tr>
</tbody>
</table>
Euclid in the i386

```
jmp .L6                      # Jump to local label .L6
.p2align 4,,7                # Skip \leq 7 bytes to a multiple of 16
.L4:
  movl %ecx,%eax
  movl %ebx,%ecx
.L6:
  cltd                      # Sign-extend eax to edx:eax
  idivl %ecx                # Compute edx:eax / ecx
  movl %edx,%ebx
  testl %edx,%edx
  jne .L4
  movl %ecx,%eax
  movl -4(%ebp),%ebx
leave
ret
```
Euclid on the i386

jmp .L6
.p2align 4,,7
.L4:
    movl %ecx,%eax   # m = n
    movl %ebx,%ecx   # n = r
.L6:
    cltd
    idivl %ecx
    movl %edx,%ebx
    testl %edx,%edx  # AND of edx and edx
    jne .L4          # branch if edx was ≠ 0
    movl %ecx,%eax   # Return n
    movl -4(%ebp),%ebx
    leave            # Move ebp to esp, pop ebp
    ret              # Pop return address and branch
SPARC Programmer’s Model

<table>
<thead>
<tr>
<th>31 0</th>
<th>31 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r24/i0</td>
</tr>
<tr>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>r24/i0</td>
</tr>
<tr>
<td>r8/o0</td>
<td>r30/i6</td>
</tr>
<tr>
<td>:</td>
<td>r31/i7</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>r14/o6</td>
<td>PSW</td>
</tr>
<tr>
<td>r15/o7</td>
<td>PC</td>
</tr>
<tr>
<td>r16/l0</td>
<td>nPC</td>
</tr>
<tr>
<td>:</td>
<td></td>
</tr>
<tr>
<td>r23/l7</td>
<td></td>
</tr>
</tbody>
</table>

- **Global Registers**: r0 to r7
- **Output Registers**: r8 to r14
- **Stack Pointer**: r15
- **Local Registers**: r16 to r23
- **Input Registers**: r24
- **Status Word**: PSW
- **Program Counter**: PC
- **Next PC**: nPC
The output registers of the calling procedure become the inputs to the called procedure. The global registers remain unchanged. The local registers are not visible across procedures.
Euclid on the SPARC

```
.file "euclid.c" # Boilerplate
gcc2.compiled.: # make .rem linker-visible
  .global .rem # Executable code
  .section ".text"
  .align 4 # make gcd linker-visible
  .global gcd
  .type gcd, #function
  .proc 04
  gcd:
  save %sp, -112, %sp # Next window, move SP
  mov %i0, %o1 # Move m into o1
  b .LL3 # Unconditional branch
  mov %i1, %i0 # Move n into i0
```
Euclid on the SPARC

```assembly
mov  %i0, %o1
b    .LL3
mov  %i1, %i0
LLL5:
    mov  %o0, %i0  # n = r
LLL3:
    mov  %o1, %o0  # Compute the remainder of
    call  .rem, 0  # m / n, result in o0
    mov  %i0, %o1
    cmp  %o0, 0
    bne  .LL5
    mov  %i0, %o1  # m = n (always executed)
    ret  # Return (actually jmp i7 + 8)
restore  # Restore previous window
```
Digital Signal Processor Apps.

Low-cost embedded systems

- Modems, cellular telephones, disk drives, printers

High-throughput applications

- Halftoning, base stations, 3-D sonar, tomography

PC based multimedia

- Compression/decompression of audio, graphics, video
Embedded Processor Requirements

Inexpensive with small area and volume

Deterministic interrupt service routine latency

Low power: $\approx 50 \text{ mW}$ (TMS320C54x uses $0.36 \mu\text{A}/\text{MIPS}$)
Conventional DSP Architecture

Harvard architecture

- Separate data memory/bus and program memory/bus
- Three reads and one or two writes per instruction cycle

Deterministic interrupt service routine latency

Multiply-accumulate in single instruction cycle

Special addressing modes supported in hardware

- Modulo addressing for circular buffers for FIR filters
- Bit-reversed addressing for fast Fourier transforms

Instructions to keep the pipeline (3-4 stages) full

- Zero-overhead looping (one pipeline flush to set up)
- Delayed branches
## Conventional DSPs

<table>
<thead>
<tr>
<th></th>
<th>Fixed-Point</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cost/Unit</strong></td>
<td>$5–$79</td>
<td>$5–$381</td>
</tr>
<tr>
<td><strong>Architecture</strong></td>
<td>Accumulator load-store</td>
<td>load-store</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>2–4 data, 8 address</td>
<td>8–16 data, 8–16 address</td>
</tr>
<tr>
<td><strong>Data Words</strong></td>
<td>16 or 24 bit</td>
<td>32 bit</td>
</tr>
<tr>
<td><strong>Chip Memory</strong></td>
<td>2–64K data+program</td>
<td>8–64K data+program</td>
</tr>
<tr>
<td><strong>Address Space</strong></td>
<td>16–128K data</td>
<td>16M–4G data</td>
</tr>
<tr>
<td></td>
<td>16–64K program</td>
<td>16M–4G program</td>
</tr>
<tr>
<td><strong>Compilers</strong></td>
<td>Bad C</td>
<td>Better C, C++</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>TI TMS320C5x</td>
<td>TI TMS320C3x</td>
</tr>
<tr>
<td></td>
<td>Motorola 56000</td>
<td>Analog Devices SHARC</td>
</tr>
</tbody>
</table>
Conventional DSPs

Market share: 95% fixed-point, 5% floating-point

Each processor comes in dozens of configurations

- Data and program memory size
- Peripherals: A/D, D/A, serial, parallel ports, timers

Drawbacks

- No byte addressing (needed for image and video)
- Limited on-chip memory
- Limited addressable memory on most fixed-point DSPs
- Non-standard C extensions to support fixed-point data
Example

Finite Impulse Response filter (FIR)
Can be used for lowpass, highpass, bandpass, etc.
Basic DSP operation
For each sample, computes

\[ y_n = \sum_{i=0}^{k} a_i x_{n+i} \]

where

\( a_0, \ldots, a_k \) are filter coefficients,
\( x_n \) is the \( n \)th input sample, \( y_n \) is the \( n \)th output sample.
56000 Programmer’s Model

Source Registers

Accumulator

Program Counter

Status Register

Loop Address

Loop Count

PC Stack

SR Stack

Stack pointer

Address Registers
56001 Memory Spaces

Three memory regions, each 64K:

- 24-bit Program memory
- 24-bit X data memory
- 24-bit Y data memory

Idea: enable simultaneous access of program, sample, and coefficient memory

Three on-chip memory spaces can be used this way

One off-chip memory pathway connected to all three memory spaces

Only one off-chip access per cycle maximum
56001 Address Generation

Addresses come from pointer register r0 . . . r7

Offset registers n0 . . . n7 can be added to pointer

Modifier registers cause the address to wrap around

Zero modifier causes reverse-carry arithmetic

<table>
<thead>
<tr>
<th>Address</th>
<th>Notation</th>
<th>Next value of r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>(r0)</td>
<td>r0</td>
</tr>
<tr>
<td>r0 + n0</td>
<td>(r0+n0)</td>
<td>r0</td>
</tr>
<tr>
<td>r0</td>
<td>(r0)+</td>
<td>(r0 + 1) mod m0</td>
</tr>
<tr>
<td>r0 - 1</td>
<td>-(r0)</td>
<td>r0 - 1 mod m0</td>
</tr>
<tr>
<td>r0</td>
<td>(r0)-</td>
<td>(r0 - 1) mod m0</td>
</tr>
<tr>
<td>r0</td>
<td>(r0)+n0</td>
<td>(r0 + n0) mod m0</td>
</tr>
<tr>
<td>r0</td>
<td>(r0)-n0</td>
<td>(r0 - n0) mod m0</td>
</tr>
</tbody>
</table>
FIR Filter in 56001

n equ 20  # Define symbolic constants
start equ $40
samples equ $0
coeffs equ $0
input equ $ffe0  # Memory-mapped I/O
output equ $ffe1

org p:start  # Locate in prog. memory
move #samples, r0  # Pointers to samples
move #coeffs, r4  # and coefficients
move #n−1, m0  # Prepare circular buffer
move m0, m4
FIR Filter in 56001

movep y:input, x:(r0)  # Load sample into memory
    # Clear accumulator A
    # Load a sample into x0
    # Load a coefficient
clr  a  x:(r0)+, x0  y:(r4)+, y0

rep  #n-1  # Repeat next instruction n-1 times
    # a = x0 × y0
    # Next sample
    # Next coefficient
mac  x0,y0,a  x:(r0)+, x0  y:(r4)+, y0
macr x0,y0,a (r0)-
movep a, y:output  # Write output sample
TI TMS320C6000 VLIW DSP

Eight instruction units dispatched by one very long instruction word

Designed for DSP applications

Orthogonal instruction set

Big, uniform register file (16 32-bit registers)

Better compiler target than 56001

Deeply pipelined (up to 15 levels)

Complicated, but more regular, datapath
Pipelining on the C6

One instruction issued per clock cycle

Very deep pipeline

- 4 fetch cycles
- 2 decode cycles
- 1-10 execute cycles

Branch in pipeline disables interrupts

Conditional instructions avoid branch-induced stalls

No hardware to protect against hazards

- Assembler or compiler’s responsibility
FIR in One ’C6 Assembly Instruction

Load a halfword (16 bits)
Do this on unit D1

FIRLOOP:

LDH .D1 *A1++, A2 ; Fetch next sample
LDH .D2 *B1++, B2 ; Fetch next coeff.
[B0] SUB .L2 B0, 1, B0 ; Decrement count
[B0] B .S2 FIRLOOP ; Branch if non-zero
MPY .M1X A2, B2, A3 ; Sample × Coeff.
ADD .L1 A4, A3, A4 ; Accumulate result

Use the cross path
Predicated instruction (only if B0 non-zero)
Run these instruction in parallel
Peripherals

Often the whole point of the system

Memory-mapped I/O

- Magical memory locations that make something happen or change on their own

Typical meanings:

- Configuration (write)
- Status (read)
- Address/Data (access more peripheral state)
**Example: 56001 Port C**

Nine pins each usable as either simple parallel I/O or as part of two serial interfaces.

**Pins:**

<table>
<thead>
<tr>
<th>Parallel</th>
<th>Serial</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC0</td>
<td>RxD</td>
<td>Serial Communication Interface (SCI)</td>
</tr>
<tr>
<td>PC1</td>
<td>TxD</td>
<td></td>
</tr>
<tr>
<td>PC2</td>
<td>SCLK</td>
<td></td>
</tr>
<tr>
<td>PC3</td>
<td>SC0</td>
<td>Synchronous Serial Interface (SSI)</td>
</tr>
<tr>
<td>PC4</td>
<td>SC1</td>
<td></td>
</tr>
<tr>
<td>PC5</td>
<td>SC2</td>
<td></td>
</tr>
<tr>
<td>PC6</td>
<td>SCK</td>
<td></td>
</tr>
<tr>
<td>PC7</td>
<td>SRD</td>
<td></td>
</tr>
<tr>
<td>PC8</td>
<td>STD</td>
<td></td>
</tr>
</tbody>
</table>
Port C Registers for Parallel Port

Port C Control Register
Selects mode (parallel or serial) of each pin
X: $FFE1 Lower 9 bits: 0 = parallel, 1 = serial

Port C Data Direction Register
I/O direction of parallel pins
X: $FFE3 Lower 9 bits: 0 = input, 1 = output

Port C Data Register
Read = parallel input data, Write = parallel data out
X: $FFE5 Lower 9 bits
Port C SCI

Three-pin interface
422 Kbit/s NRZ asynchronous interface (RS-232-like)
3.375 Mbit/s synchronous serial mode
Multidrop mode for multiprocessor systems
Two Wakeup modes
  • Idle line
  • Address bit
Wired-OR mode
On-chip or external baud rate generator
Four interrupt priority levels
Port C SCI Registers

SCI Control Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–2</td>
<td>Word select bits</td>
</tr>
<tr>
<td>3</td>
<td>Shift direction</td>
</tr>
<tr>
<td>4</td>
<td>Send break</td>
</tr>
<tr>
<td>5</td>
<td>Wakeup mode select</td>
</tr>
<tr>
<td>6</td>
<td>Receiver wakeup enable</td>
</tr>
<tr>
<td>7</td>
<td>Wired-OR mode select</td>
</tr>
<tr>
<td>8</td>
<td>Receiver enable</td>
</tr>
<tr>
<td>9</td>
<td>Transmitter enable</td>
</tr>
<tr>
<td>10</td>
<td>Idle line interrupt enable</td>
</tr>
<tr>
<td>11</td>
<td>Receive interrupt enable</td>
</tr>
<tr>
<td>12</td>
<td>Transmit interrupt enable</td>
</tr>
<tr>
<td>13</td>
<td>Timer interrupt enable</td>
</tr>
<tr>
<td>15</td>
<td>Clock polarity</td>
</tr>
</tbody>
</table>
Port C SCI Registers

SCI Status Register  (Read only)

<table>
<thead>
<tr>
<th>X: $FFF1</th>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Transmitter Empty</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Transmitter Reg Empty</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Receive Data Full</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Idle Line</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Overrun Error</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Parity Error</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Framing Error</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Received bit 8</td>
</tr>
</tbody>
</table>
Port C SCI Registers

### SCI Clock Control Register

<table>
<thead>
<tr>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>11–0</td>
<td>Clock Divider</td>
</tr>
<tr>
<td>12</td>
<td>Clock Output Divider</td>
</tr>
<tr>
<td>13</td>
<td>Clock Prescaler</td>
</tr>
<tr>
<td>14</td>
<td>Receive Clock Source</td>
</tr>
<tr>
<td>15</td>
<td>Transmit Clock Source</td>
</tr>
</tbody>
</table>
Port C SSI

Intended for synchronous, constant-rate protocols
Easy interface to serial ADCs and DACs
Many more operating modes than SCI
Six Pins (Rx, Tx, Clk, Rx Clk, Frame Sync, Tx Clk)
8, 12, 16, or 24-bit words
Port C SSI Registers

SSI Control Register A $FFEC
Prescaler, frame rate, word length

SSI Control Register B $FFED
Interrupt enables, various mode settings

SSI Status/Time Slot Register $FFEE
Sync, empty, oerrun

SSI Receive/Transmit Data Register $FFEF
8, 16, or 24 bits of read/write data.