Assembly Languages I

Prof. Stephen A. Edwards

Last Time

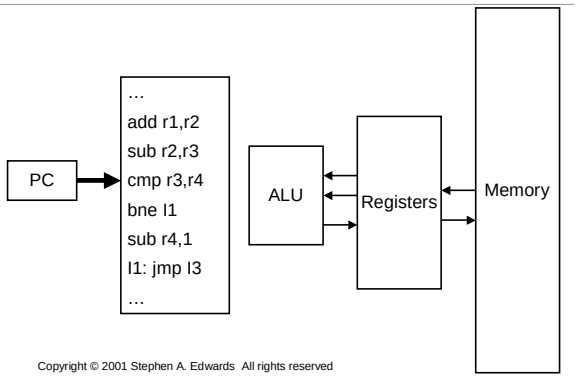
- **Languages**
- **Syntax: what's in the language**
- **Semantics: what the language means**
- **Model: what the language manipulates**
- **Specification asks for something**
- **Modeling asks what something will do**
- **Concurrency**
- **Nondeterminism**

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**

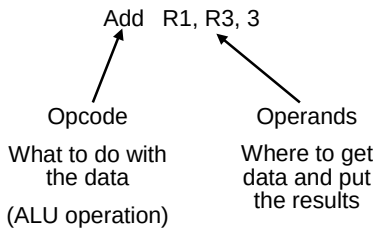
ENIAC, 1946
17k tubes, 5kHz

Assembly Language Model



Assembly Language Instructions

- **Built from two pieces**



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)

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

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

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

Types of Assembly Languages

	CISC	RISC	DSP	VLIW
Opcodes	Many, Complex	Few, Simple	Few, Complex	Few, Simple
Registers	Few, Special	Many, General	Few, Special	Many, General
Addressing modes	Many	Few	Special	Few
Instruction-level Parallelism	None	None	Restricted	Plenty

Gratuitous Picture

- **Woolworth building**
- **Cass Gilbert, 1913**
- **Application of the Gothic style to a 792' skyscraper**
- **Tallest building in the world when it was constructed**
- **Downtown: near City Hall**

Example: Euclid's Algorithm

- **In C:**

```
int gcd(int m, int n)
{
  int r;
  while ( (r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

Two integer parameters

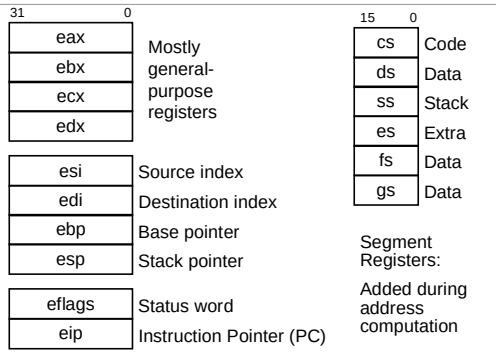
One local variable

Remainder operation

Non-zero test

Data transfer

i386 Programmer's Model



Euclid's Algorithm 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
```

Boilerplate

Assembler directives start with "."

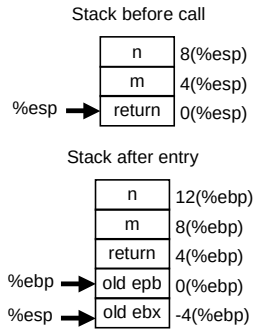
"This will be executable code"

"Start on a 16-byte boundary"

"gcd is a linker-visible label"

Euclid's Algorithm 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
```



Euclid's Algorithm on the i386

```
    jmp .L6
    .p2align 4,,7
.L4:
    movl %ecx,%eax
    movl %ebx,%ecx
.L6:
    cltd
    idivl %ecx
    movl %edx,%ebx
    testl %edx,%edx
    jne .L4
    movl %ecx,%eax
    movl -4(%ebp),%ebx
    leave
    ret
```

"Jump to local label .L6"

"Skip as many as 7 bytes to start on a 16-byte boundary"

"Sign-extend %eax to %edx:%eax"

"Compute %edx:%eax ÷ %ecx: quotient in %eax, remainder in %edx"

Register assignments:

%eax	m
%ebx	r
%ecx	n

```
while ((r = m % n) != 0) {
  m = n;
  n = r;
}
return n;
```


Euclid's Algorithm on the i386

```
    jmp .L6
    .p2align 4,,7
.L4:
    movl %ecx,%eax
    movl %ebx,%ecx
.L6:
    cltd
    idivl %ecx
    movl %edx,%ebx
    testl %edx,%edx
    jne .L4
    movl %ecx,%eax
    movl -4(%ebp),%ebx
    leave
    ret
```

"m = n"

"n = r"

"compute AND of %edx and %edx, update the status word, discard the result"

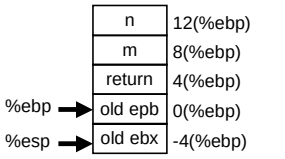
"Branch back to .L4 if the zero flag is clear, I.e., the last arithmetic operation did not produce zero"

```
while ((r = m % n) != 0) {
  m = n;
  n = r;
}
return n;
```


Euclid's Algorithm on the i386

```
    jmp .L6
    .p2align 4,,7
.L4:
    movl %ecx,%eax
    movl %ebx,%ecx
.L6:
    cltd
    idivl %ecx
    movl %edx,%ebx
    testl %edx,%edx
    jne .L4
    movl %ecx,%eax
    movl -4(%ebp),%ebx
    leave
    ret
```

Stack before exit



"return n" (caller expects value in %eax)

"move %ebp to %esp and pop %ebp from the stack"

"Pop a return address from the stack and branch to it"

Another Gratuitous Picture

- **Types of Bridges**

Suspension

(Golden Gate Bridge, California)

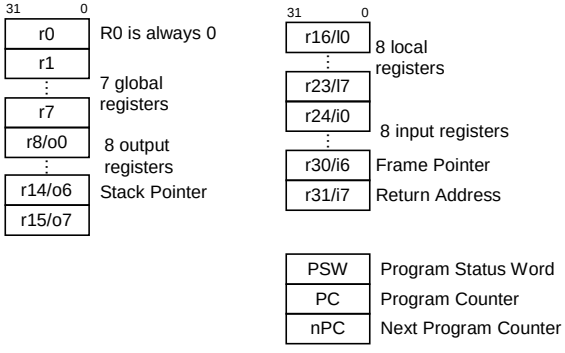
Truss

(Forth Bridge, Scotland)

Cable-stayed

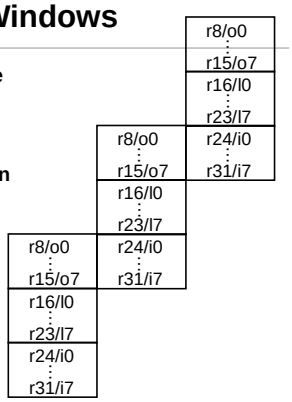
(Higashi Kobe, Japan)

SPARC Programmer's Model



SPARC Register Windows

- **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's Algorithm on the SPARC

```
        .file   "euclid.c"
gcc2_compiled.:
        .global .rem
.section        ".text"
        .align 4
        .global gcd
        .type   gcd,#function
        .proc   04
gcd:
        save    %sp, -112, %sp

        mov     %i0, %o1
        b       .LL3
        mov     %i1, %i0
```

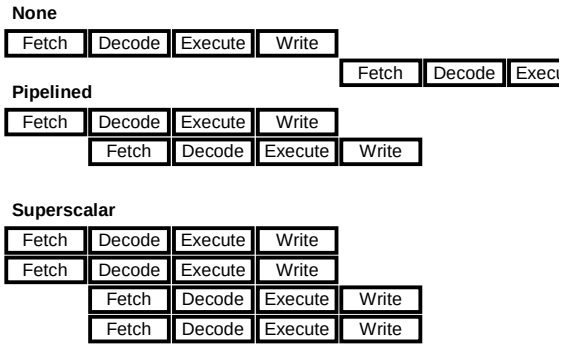
Boilerplate

Assembler directives start with "."

"This will be executable code"

"gcd is a linker-visible label"

Pipelining



Euclid's Algorithm on the SPARC

```
        .file   "euclid.c"
gcc2_compiled.:
        .global .rem
.section        ".text"
        .align 4
        .global gcd
        .type   gcd,#function
        .proc   04
gcd:
        save    %sp, -112, %sp

        mov     %i0, %o1
        b       .LL3
        mov     %i1, %i0
```

"Advance the register windows. Allocate space on the stack."

"Move argument 0 (m) into %o1"

"Branch to .LL3 after executing the next instruction"

The SPARC doesn't have a mov instruction: the assembler replaces this with

or %g0, %i1, %i0

Euclid's Algorithm on the SPARC

```
        mov     %i0, %o1
        b       .LL3
        mov     %i1, %i0
.LL5:
        mov     %o0, %i0
.LL3:
        mov     %o1, %o0
        call    .rem, 0
        mov     %i0, %o1
        cmp     %o0, 0
        bne     .LL5
        mov     %i0, %o1
        ret
        restore
```

```
while ((r = m % n) != 0) {
  m = n;
  n = r;
}
return n;
```

"Compute the remainder of m ÷ n (result in %o0)"

Call is also delayed

Register assignments:

m	%o1
r	%o0
n	%i0

Euclid's Algorithm on the SPARC

```
        mov     %i0, %o1
        b       .LL3
        mov     %i1, %i0
.LL5:
        mov     %o0, %i0
.LL3:
        mov     %o1, %o0
        call    .rem, 0
        mov     %i0, %o1
        cmp     %o0, 0
        bne     .LL5
        mov     %i0, %o1
        ret
        restore
```

```
while ((r = m % n) != 0) {
  m = n;
  n = r;
}
return n;
```

Register assignments:

m	%o1
r	%o0
n	%i0

"n = r"

"m = n" (executed even if loop terminates)

"Branch back to caller"
SPARC has no ret: this is jmp %i7 + 8

Inverse of save: return to previous register window