Introduction to Assembly Code

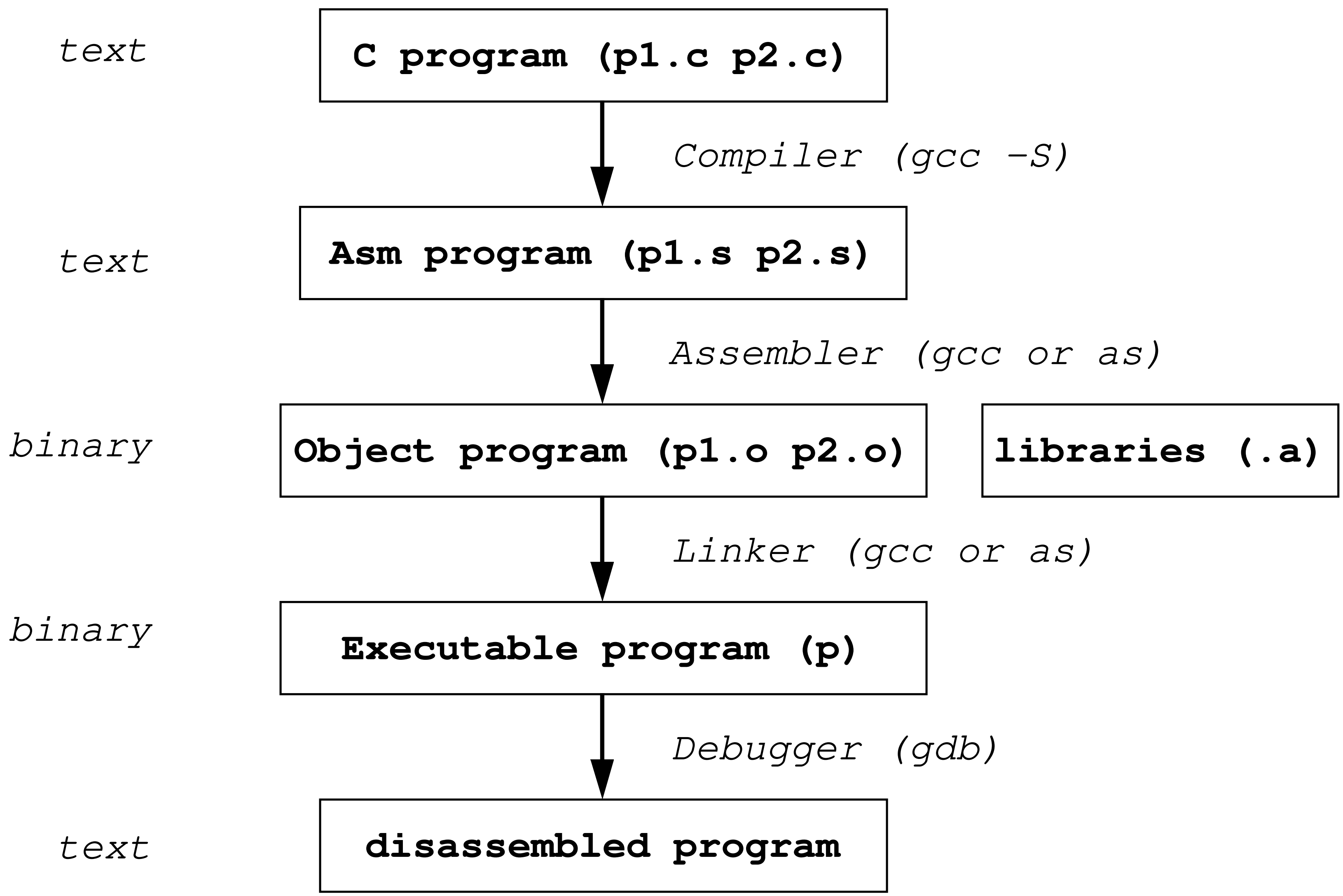
CS 347

Jan. 18, 1996

Topics

- Program representations
- Compiler generated assembly
- Register utilization
- Delayed branches
- Pointers and arrays
- Structure indexing

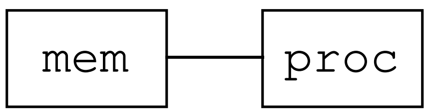
Translation Process



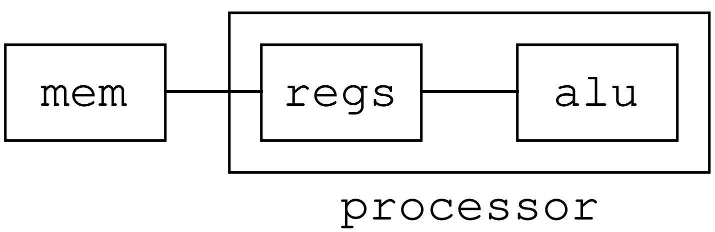
Abstract machines

machine model	data	control
C	1) char 2) int float 3) long double 4) struct array 5) pointer	1) loops 2) conditionals 3) goto
ASM	1) byte 2) word 3) doubleword 4) contiguous word allocation 5) word byte address	3) goto

C



ASM



Translation Tasks

Compiler

- syntax and type checking
- storage allocation
- register assignment and code generation

Assembler

- encode instructions
- resolve local addresses
- rearrange instructions (especially with MIPS)

Linker

- combine object files
- resolve global addresses
- create executable image for loader

MIPS Register Convention

Integer Registers

- 32 total
- Fast access to program data

Conventions

- Established as part of architecture
- Used by all compilers, programs, and libraries
- Assures compatibility

`$0`	`$0`	Constant 0
`$1`	`$at`	Reserved Temp.
`$2`	`$v0`	Return Values
`$3`	`$v1`	
`$4`	`$a0`	Procedure arguments
`$5`	`$a1`	
`$6`	`$a2`	
`$7`	`$a3`	
`$8`	`$t0`	Caller Save Temporaries: May be overwritten by called procedures
`$9`	`$t1`	
`$10`	`$t2`	
`$11`	`$t3`	
`$12`	`$t4`	
`$13`	`$t5`	
`$14`	`$t6`	
`$15`	`$t7`	

Registers (cont.)

Important Ones for Now

\$0	Constant 0
\$2	Return Value
\$3	Can use as temporary
\$4	First argument
\$5	Second argument
\$31	Return address

\$16	`$s0`	Callee Save Temporaries: May not be overwritten by called procedures
\$17	`$s1`	
\$18	`$s2`	
\$19	`$s3`	
\$20	`$s4`	
\$21	`$s5`	
\$22	`$s6`	
\$23	`$s7`	
\$24	`$t8`	Caller Save Temp
\$25	`$t9`	
\$26	`$k0`	Reserved for Operating Sys
\$27	`$k1`	
\$28	`$gp`	Global Pointer
\$29	`$sp`	Stack Pointer
\$30	`$s8`	Callee Save Temp
\$31	`$ra`	Return Address

Program Representations

C Code

```
void test1(int x, int y)
{
  gval = (x+x+x) - (y+y+y);
}
```

Compiled to Assembly

```
        .loc    1 7
        .ent    test1
test1:
        .frame  $sp,0,$31
        .mask   0x00000000,0
        .fmask  0x00000000,0
        addu    $3,$4,$4
        addu    $3,$3,$4
        addu    $2,$5,$5
        addu    $2,$2,$5
        subu    $3,$3,$2
        sw      $3,gval
        j       $31
        .end    test1
```

Prog. Representation (Cont.)

Object

```
0x400230 <test1>:
    0x00841821
    0x00a51021
    0x00451021
    0x00641821
    0x00621823
    0x03e00008
    0xaf838108
```

Disassembled

```
0x400230 <test1>:       addu $v1,$a0,$a0
0x400234 <test1+4>:     addu $v0,$a1,$a1
0x400238 <test1+8>:     addu $v0,$v0,$a1
0x40023c <test1+12>:    addu $v1,$v1,$a0
0x400240 <test1+16>:    subu $v1,$v1,$v0
0x400244 <test1+20>:    jr $ra
0x400248 <test1+24>:    sw $v1,-32504($gp)
```

Important Lessons:

Data and code are just bit patterns.

The meaning of a bit pattern depends on its context.

Understanding Assembly

C Code

```
void test1(int x, int y)
{
  gval = (x+x+x) - (y+y+y);
}
```

Annotated Assembly

```
test1:
       # x in $4, y in $5
        addu    $3,$4,$4    # $3 = x+x
        addu    $3,$3,$4    # $3 += x
        addu    $2,$5,$5    # $2 = y+y
        addu    $2,$2,$5    # $2 += y
        subu    $3,$3,$2    # $3 -= $2
        sw      $3,gval     # Store diff.
        j       $31         # Return
```

Delayed Branches

C Code

```c
int test2(int x, int y)
{
  return (x+x+x) - (y+y+y);
}
```

Compiled to Assembly

```
        .loc     1 12
        .ent     test2
test2:
        .frame   $sp,0,$31
        .mask    0x00000000,0
        .fmask   0x00000000,0
        addu     $2,$4,$4
        addu     $2,$2,$4
        addu     $3,$5,$5
        addu     $3,$3,$5
        .set     noreorder
        .set     nomacro
        j        $31
        subu     $2,$2,$3
        .set     macro
        .set     reorder

        .end     test2
```

Annotated Delayed Branch Code

C Code

```
int test2(int x, int y)
{
  return (x+x+x) - (y+y+y);
}
```

Annotated Assembly

```
test2:
        # x in $4, y in $5
        # Return result in $2
        addu    $2,$4,$4   # $2 = x+x
        addu    $2,$2,$4   # $2 += x
        addu    $3,$5,$5   # $3 = y+y
        addu    $3,$3,$5   # $3 += y
        J       $31        # (Delayed) return
        subu    $2,$2,$3   # with value $2-$3
```

Our Notational Convention

- Uppercase mnemonic to indicate delayed branch
- Lowercase to indicate normal
- Not legal MIPS code

Pointer Examples

C Code

```
int iaddp(int *xp, int *yp)
{
  int x = *xp;
  int y = *yp;
  return x + y;
}
```

```
void incr(int *sum, int v)
{
  int old = *sum;
  int new = old+val;
  *sum = new;
}
```

Annotated Assembly

```
iaddp:
        lw      $2,0($4)   # $2 = *xp
        lw      $3,0($5)   # $3 = *yp
        J       $31        # (delayed) return
        addu    $2,$2,$3   # with value $2+$3
```

```
incr:
        lw      $2,0($4)   # $2 = *sum
        #nop
        addu    $2,$2,$5   # $2+= v
        J       $31        # (delayed) return
        sw      $2,0($4)   # with *sum = $2
```

Array Indexing

C Code

```
int aref(int a[], int i)
{
  return a[i];
}
```

```
int garray[10];

int gref(int i)
{
  return garray[i];
}
```

Annotated Assembly

```
aref:
        sll     $5,$5,2  # $5 = 4*i
        addu    $5,$5,$4 # $5 = &a[i]
        lw      $2,0($5) # return val. = a[i]
        j       $31      # Return
```

```
        .comm   garray,40

gref:
    sll $4,$4,2        # $4 = 4*i
    lw  $2,garray($4)  #Return val = garray[i]
    j   $31            # Return
```

Disassembled:

```
0x40034c <gref>:        sll $a0,$a0,0x2
0x400350 <gref+4>:      lui $v0,4096
0x400354 <gref+8>:      addu $v0,$v0,$a0
0x400358 <gref+12>:     lw $v0,4304($v0)
0x40035c <gref+16>:     JR $ra
0x400360 <gref+20>:     nop
```

Structures & Pointers

C Code

```
struct rec {
  int i;
  int a[3];
  int *p;
};
```

i	a	p
0	4	16

```
void addr(struct rec *r)
{
  int *loc;
  r->i = 1;
  loc = &r->a[r->i];
  r->p = loc;
  *(r->p) = 2;
  r->a[0] = 4;
  *(r->p+1) = 8;
}
```

Annotated Assembly

```
addr:
        li      $2,0x00000001 # $2 = 1
        sw      $2,0($4)      # r->i = 1
        addu    $2,$4,8       # $2 = &r->a[1]
        sw      $2,16($4)     # r->p = $2
        li      $2,0x00000002 # $2 = 1
        sw      $2,8($4)      # r->a[1] = 2
        lw      $2,16($4)     # $2 = r->p
        li      $3,0x00000004 # $3 = 4
        li      $5,0x00000008 # $4 = 8
        sw      $3,4($4)      # r->a[0] = 4
        J       $31           # Return
        sw      $5,4($2)      # setting
                              # *(r->p+1) = 8
```