Compiler Design and Construction
Code Generation
Pop Quiz/Review

- What options do we have for generating code?

- If we choose IR, what options do we have for IR?
Intermediate Code Generation

A OR B AND NOT C

t1 = not C

t2 = B AND t1

t3 = A OR t2
Intermediate Code Generation

If $A < B$ AND $C < D$

$$A = C \times 10 + D$$

$t1 = A < B$
$t2 = C < D$
$t3 = t1$ AND $t2$
$t3$ goto true1
goto endif1
true1: $t4 = c*10$
$t5 = t4 \times D$
$A = t5$
endif1:
While \( a < b \) do
  if \( c < d \) then \( x = y + 2 \)
  else \( x = y - 2 \)

0 \( t_1 = A < B \)
1 \( t_1 \) goto 3
2 goto 11
3 \( t_2 = c < d \)
4 \( t_2 \) goto 8
5 \( t_3 = y - 2 \)
6 \( x = t_3 \)
7 goto 10
8 \( t_4 = y + 2 \)
9 \( x = t_4 \)
10 goto 0
Generating Code via Macro Expansion

- Macroexpand each IR tuple or subtree
  
  \((\text{ADDI}, \text{Addr}(b), \text{Addr}(c), t1)\)

  \(\text{lw } \$t0, b\)

  \(\text{lw } \$t1, c\)

  \(\text{add } \$t2, \$t0, \$t1\)

- Macroexpansion gives poor quality code if each tuple expanded separately
  
  - Ignoring state (values already loaded)
Generating Code via Macro Expansion

- A := B+C;
- D := A * C;

lw $t0, B,
lw $t1, C,
add $t2, $t0, $t1
sw $t2, A
lw $t0, A
lw $t1, C
mul $t2, $t0, $t1
sw $t2, D
Generating Code via Macro Expansion

- D := (B+C)*C;

\[ t1 = B + C; \]
\[ lw \ $t0, B, \]
\[ lw \ $t1, C \]
\[ add \ $t2, \ $t0, \ $t1 \]
\[ sw \ $t2, t1 \]

\[ t2 = t1 * C; \]
\[ lw \ $t0, t1 \]
\[ lw \ $t1, C \]
\[ mul \ $t2, \ $t0, \ $t1 \]
\[ sw \ $t2, t2 \]

\[ d = t2 \]
\[ lw \ $t0, t2 \]
\[ sw \ $t0, d \]
Generating Code via Macro Expansion

- Macroexpansion gives poor quality code if each tuple expanded separately
  - Ignoring state (values already loaded)

- What if more than 1 tuple can be replaced with 1 instruction
  - Powerful addressing modes
  - Powerful instructions
  - Loop construct that decrements, tests and jumps if necessary
Register and Temporary Management

- Efficient use of registers
  - Values used often remain in registers
  - Temp values reused if possible

- Define Register classes
  - Allocatable
    - Explicitly allocated and freed
  - Reserved
    - Have a fixed function
  - Volatile
    - Can be used at any time
    - Good for temp values (A:=B)
Temporaries

- Usually in registers (for quick access)

- Storage temporaries
  - Reside in memory
  - Save registers or large data objects

- Pseudoregisters
  - Load into volatile, then save back out
  - Generates poor code
  - Moving temp from register to pseudo register is called spilling
Code Generation

- A separate generator for each tuple
  - Modularized
  - Simpler
  - Harder to generate good code
  - Easy to add to yacc!
- A single generator
  - More complex
Code Generation

- Instruction selection
  - Addressing modes, intermediates
  - R-R, R-M, M-M, RI...

- Address-mode selection
  - Remember all the types!

- Register allocation

- These are tightly coupled
  - Address-mode affects instruction
  - Instruction can affect register

- See handout for a “+” code generator
  - Doesn't handle 0 or same oprnd twice
Expressions in YACC

expression : operand mathop operand {
    if CheckType($2, $1, $3) yyerror(“operand mismatch”);
    emit($2, $1, $3);
}

operand: INTCONST {<$e.type>$ = TY_INT; }
    | FLCONST {<$e.type>$ = TY_FLT;}
    | ID {SYMTAB *p = symLookUp($1);
        if (p) {emit(lw)??}
        else yyerror2(“error: %s undefined”, $1);"}
Code Generation

- IF A < B THEN thenPart ELSE elsePart END IF;

  blt $t0, $t1, _then
  j _else:
  _then: thenPart
  j _endif;
  _else: elsePart
  _endif:
Code Generation

- IF A < B THEN thenPart ELSE elsePart END IF;

  _then24:  bge $t0, $t1, _else
  thenPart
  _endif:

  _else:  blt $t0, $t1,
  _then:  _then:
  j _endif;
  thenPart

  _else:  j _else:
  _endif:
  elsePart

  _endif:
IF A < B THEN A := C * 10;

```
lw $t0, A
lw $t1, B
bge $t0, $t1, _else24
lw $t2, C
li $t3, 10
mul $t4, $t2, $t3
sw $t4, A
```
Code Generation

IF A < B THEN A := C * 10; ELSE A := C*9; END IF;

lw $t0, A
lw $t1, B
bge $t0, $t1, _else24
lw $t2, C
li $t3, 10
mul $t4, $t2, $t3
sw $t4, A
j _endif24

_else24:
lw $t5, C
li $t6, 9
mul $t7, $t6, $t6
sw $t7, A

_endif24:
Decls: type varlist SEMI | type varlist SEMI decls |
varlist: VAR {
    addST($1,$0);
    printf("%s: %s",$1,($0 == 1)? ".float 0.0": ".word 0");
}
| varlist COMMA VARIABLE {
    addST($3,$0);
    printf("%s: %s",$3,($0 == 1)? ".float 0.0": ".word 0");
}
type: FLOAT
    {$$ = TY_FLT; }
| INTEGER
    {$$ = TY_INT; }
;
A Complete Example

PROGRAM
  INTEGER B[15];
BEGIN
  B[8] := 19;
END

.data
B: .word 0:15
.text
.globl main
main:
  li $t0, 8
  li $t1, 19
  la $t2, B # array base address
  mul $t3, $t0, 4 # offset to element
  add $t2, $t2, $t3 # address of element
  sw $t1, 0($t2) # save rhs in lhs array
  li $v0, 10
  syscall # exit
A Digression into MIPS32
 Registers (MIPS)

- **32 registers provided (but not 32-useable registers!)**
  - R0 .. R31
  - Register R0 is hard-wired to zero
  - Register R1 is reserved for assembler
- **Arithmetic instructions operands must be registers**
MIPS: Software conventions for Registers

- Registers all have two names, ie $3$ and $v1$
- Although you can do what you want, you should follow these conventions:

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero; constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at; reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0; expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1; function results</td>
</tr>
<tr>
<td>4</td>
<td>a0; arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0; temporary: caller saves</td>
</tr>
<tr>
<td></td>
<td>(callee can clobber)</td>
</tr>
<tr>
<td>15</td>
<td>t7</td>
</tr>
<tr>
<td>16</td>
<td>s0; local variables</td>
</tr>
<tr>
<td></td>
<td>... (callee must save)</td>
</tr>
<tr>
<td>23</td>
<td>s7</td>
</tr>
<tr>
<td>24</td>
<td>t8; temporary (cont’d)</td>
</tr>
<tr>
<td>25</td>
<td>t9</td>
</tr>
<tr>
<td>26</td>
<td>k0; reserved for OS kernel</td>
</tr>
<tr>
<td>27</td>
<td>k1</td>
</tr>
<tr>
<td>28</td>
<td>gp; Pointer to global area</td>
</tr>
<tr>
<td>29</td>
<td>sp; Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>fp; frame pointer</td>
</tr>
<tr>
<td>31</td>
<td>ra; Return Address (HW)</td>
</tr>
</tbody>
</table>
Addressing Objects: Endianess and Alignment

- **Big Endian:** address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian:** address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

Alignment: require that objects fall on address that is multiple of their size.

```
big endian byte 0
msb  3  2  1  0
  0  1  2  3
little endian byte 0
  0  1  2  3
```

```
Aligned
```

```
Not Aligned
```

25  Xiaoyu Zhang, CSUSM  CS 331
Memory Instructions

- MIPS is CISC so only load and store instructions
  - lw $t1, offset($t0);
  - sw $t1, offset($t0);

- Example:
    assume \( h \) in \$s2 and base address of the array \( A \) in \$s3
    
  - MIPS code: 
    - lw $t0, 32($s3)
    - add $t0, $s2, $t0
    - sw $t0, 32($s3)

- Store word has destination last

- Remember arithmetic operands are **registers**, not memory!
## I/O Services

<table>
<thead>
<tr>
<th>Service</th>
<th>$v0</th>
<th>Argument(s)</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Print integer</td>
<td>1</td>
<td>$a0 = number to be printed</td>
<td></td>
</tr>
<tr>
<td>Print float</td>
<td>2</td>
<td>$f12 = number to be printed</td>
<td></td>
</tr>
<tr>
<td>Print double</td>
<td>3</td>
<td>$f12 = number to be printed</td>
<td></td>
</tr>
<tr>
<td>Print string</td>
<td>4</td>
<td>$a0 = address of string in memory</td>
<td></td>
</tr>
<tr>
<td>Read integer</td>
<td>5</td>
<td></td>
<td>number returned in $v0</td>
</tr>
<tr>
<td>Read float</td>
<td>6</td>
<td></td>
<td>number returned in $f0</td>
</tr>
<tr>
<td>Read double</td>
<td>7</td>
<td></td>
<td>number returned in $f0</td>
</tr>
<tr>
<td>Read string</td>
<td>8</td>
<td>$a0 = address of input buffer in memory</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$a1 = length of buffer (n)</td>
<td></td>
</tr>
<tr>
<td>Sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>address in $v0</td>
</tr>
<tr>
<td>Exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print character</td>
<td>11</td>
<td>$a0 = character to print</td>
<td></td>
</tr>
<tr>
<td>Read character</td>
<td>12</td>
<td></td>
<td>character read in $v0</td>
</tr>
<tr>
<td>File I/O operations</td>
<td>13 – 16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exit2 (terminate with value)</td>
<td>17</td>
<td>$a0 = termination result</td>
<td></td>
</tr>
</tbody>
</table>

```assembly
li $v0, 4  # system call print_str
la $a0, str # addr of string to print
syscall    # print the string

li $v0, 1  # system call print_int
li $a0, 5  # integer to print
syscall    # print it
```
## Hello World Assembly Program

### Data Segment

```plaintext
.data
str: .asciiz "Hello world!!!\n"
```

### Text Segment (Code)

```plaintext
.text
.globl main  # exports symbol "main"
main:
    la $a0,str  # put string address into a0
    li $v0,4    # system call to print
    syscall    # out a string
    li $v0,10   # exit with no result
    syscall    # exit with no result
```

### Labels

- `str`
- `main`

### Directives

- `.data`
- `.text`
- `.globl`
Hello World Assembly Program

.data
str: .asciiz "Hello world!!!\n"

.text
.globl main  # exports symbol “main”
main:
la $a0,str  # put string address into a0
li $v0,4  # system call to print
syscall  # out a string

li $v0,10  # exit with no result
syscall  # exit with no result
Some Useful Mips Commands

- Some register-register math commands

\[
\begin{align*}
\text{add} & \: & $t0, & \: & $t1, & \: & $t2 & \# & \: & $t0 = $t1+$t2 \\
\text{sub} & \: & $t0, & \: & $t1, & \: & $t2 & \# & \: & $t0 = $t1-$t2 \\
\text{mul} & \: & $t0, & \: & $t1, & \: & $t2 & \# & \: & $t0 = $t1*$t2 \text{ note there could be overflow!} \\
\text{div} & \: & $t0, & \: & $t1, & \: & $t2 & \# & \: & $t0 = $t1/$t2 \text{ note there could be overflow!}
\end{align*}
\]

- CISC machine, so can only access memory with load/store commands

\[
\begin{align*}
\text{lw} & \: & $t1, & \: & \text{a_addr} & \# & \: & $t1 = \text{Mem}[\text{a_addr}] \\
\text{lw} & \: & $s1, & \: & 8($s0) & \# & \: & $s1 = \text{Mem}[\$s0+8] \text{ sw } $t1, \: \text{a_addr} & \# & \: & \text{Mem}[\text{a_addr}] = $t1
\end{align*}
\]

- Sometimes you need an address

\[
\begin{align*}
\text{la} & \: & $a0, & \: & \text{addr} & \# & \: & \text{put addresses addr into } $a0
\end{align*}
\]
Some Useful Mips Commands

- Some register-register math commands
  - Note CMD Destination Operand1 Operand2

  - `add $t0, $t1, $t2`  # $t0 = $t1+$t2
  - `sub $t0, $t1, $t2`  # $t0 = $t1-$t2
  - `mul $t0, $t1, $t2`  # $t0 = $t1*$t2 note there could be overflow!
  - `div $t0, $t1, $t2`  # $t0 = $t1/$t2 note there could be overflow!

- CISC machine, so can only access memory with load/store commands

  - `lw $t1, a_addr`  # $t1 = Mem[a_addr]
  - `lw $s1, 8($s0)`  # $s1 = Mem[$s0+8]
  - `sw $t1, a_addr`  # Mem[a_addr] = $t1

- Sometimes you need an address

  - `la $a0, addr`  # put addresss addr into $a0
Some Useful Mips Commands

- Those pesky immediates (constants)

li $a0, 12  # put immediate value of 12 into register $a0
mfhi $t0  # move contents from hi into $t0
Some Useful Mips Commands

- Branching

  beqz $s0, label  # if $s0 == 0 goto label
  bnez $s0, label  # if $s0 != 0 goto label
  bgez $s0, label  # if $s0 >= 0 goto label
  bge $t0, $t1, label  # if $t0 >= $t1 goto label pseudoinstruction
  bgt $t0, $t1, label  # if $t0 > $t1 goto label pseudoinstruction
  ble $t0, $t1, label  # if $t0 <= $t1 goto label pseudoinstruction
  blt $t0, $t1, label  # if $t0 < $t1 goto label pseudoinstruction
  beq $t0, $t1, label  # if $t0 == $t1 goto label
Spim, xspim, QtSpim
Use Appendix A as a Reference
Use of Registers

Example:

\[ a = (b + c) - (d + e) \]; // C statement

\# $s0 - $s4 : a - e

- add \( \text{\$t0, \$s1, \$s2} \)
- add \( \text{\$t1, \$s3, \$s4} \)
- sub \( \text{\$s0, \$t0, \$t1} \)

\[ a = b + A[4]; \] // add an array element to a var

// $s3 has address A

- lw \( \text{\$t0, 16($s3)} \)
- add \( \text{\$s1, \$s2, \$t0} \)
load and store

Ex:

\[ a = b + A[i]; \quad \text{// A is in } s3, a, b, i \text{ in } s1, s2, s4 \]

\begin{align*}
\text{add} & \quad t1, s4, s4 & \quad \# t1 = 2 \times i \\
\text{add} & \quad t1, t1, t1 & \quad \# t1 = 4 \times i \\
\text{add} & \quad t1, t1, s3 & \quad \# t1 = \text{addr. of } A[i] \\
\text{lw} & \quad t0, 0(t1) & \quad \# t0 = A[i] \\
\text{add} & \quad s1, s2, t0 & \quad \# a = b + A[i]
\end{align*}
Making Decisions

- **Example**

```c
if ( a != b) goto L1;  // x,y,z,a,b mapped to $s0-$s4
    x = y + z;
L1 : x = x - a;
```

```asm
bne $s3, $s4, L1  # goto L1 if a != b
    add $s0, $s1, $s2  # x = y + z (ignored if a!=b)
L1:sub $s0, $s0, $s3  # x = x - a (always ex)
```

- **Reminder**
  - Registers variable in C code: $s0 ... $s7 $16 ... 23
  - Registers temporary variable: $t0 ... $t7 $8 ... 15
  - Register: $zero always 0
if-then-else

Example:

if ( a==b) x = y + z;
else x = y – z ;

bne $s3, $s4, Else
add $s0, $s1, $s2
j Exit

Else : sub $s0,$s1,$s2

Exit : # goto Else if a!=b
# x = y + z
# goto Exit
# x = y – z
Example: Loop with array index

- Loop: $g = g + A[i]$;
  
  $i = i + j$;

  if ($i \neq h$) goto Loop

  ...

- $s_1, s_2, s_3, s_4 = g, h, i, j$, array $A$ base = $s_5$

**LOOP:**

```
add $t1, $s3, $s3  #$t1 = 2 * i
add $t1, $t1, $t1  #$t1 = 4 * i
add $t1, $t1, $s5  #$t1 = adr. Of A[i]
lw  $t0, 0($t1)   #load A[i]
add $s1, $s1, $t0  #g = g + A[i]
add $s3, $s3, $s4  #i = i + j
bne $s3, $s2, LOOP
```
Loops

Example:

while ( A[i] == k ) // i, j, k in $s3, $s4, $s5
    i = i + j; // A is in $s6

Loop: sll $t1, $s3, 2  # $t1 = 4 * i
    add $t1, $t1, $s6  # $t1 = addr. of A[i]
    lw $t0, 0($t1)    # $t0 = A[i]
    bne $t0, $s5, Exit # goto Exit if A[i]! = k
    add $s3, $s3, $s4 # i = i + j
    j Loop            # goto Loop

Exit:
Other decisions

- **Set R1 on R2 less than R3**
  
  \[ \text{slt R1, R2, R3} \]
  
  Compares two registers, R2 and R3
  
  \[ R1 = 1 \quad \text{if R2 < R3} \quad \text{else} \quad R1 = 0 \quad \text{if R2 >= R3} \]

- **Example**
  
  \[ \text{slt } $t1, $s1, $s2 \]

- **Branch less than**
  
  Example: \[ \text{if (A < B) goto LESS} \]
  
  \[ \text{slt } $t1, $s1, $s2 \quad \# t1 = 1 \text{ if } A < B \]
  
  \[ \text{bne } $t1, $0, LESS \]
The switch statement can be converted into a big chain of if-then-else statements.

A more efficient method is to use a jump address table of addresses of alternative instruction sequences and the `jr` instruction. Assume the table base address in `$t4`
Switch cont.

```
slt  $t3, $s5, $zero  # is k < 0
bne  $t3, $zero, Exit # if k < 0, goto Exit
slt  $t3, $s5, $t2    # is k < 4, here $t2=4
beq  $t3, $zero, Exit # if k >=4 goto Exit
sll  $t1, $s5, 2     # $t1 = 4 * k
add  $t1, $t1, $t4    # $t1 = addr. Of $t4[k]
lw   $t0, 0($t1)      # $t0 = $t4[k]
jr   $t0              # jump to addr. In $t0

# $t4[0]=&L0, $t4[1]=&L1, ...
L0  :  add  $s0, $s3, $s4  # f = i + j
     j     Exit
L1  :  add  $s0, $s1, $s2  # f = g + h
     j     Exit
L2  :  sub  $s0, $s1, $s2  # f = g – h
     j     Exit
L3  :  sub  $s0, $s1, $s2  # f = i – j
Exit :
```

# $t4[0]=&L0, $t4[1]=&L1, ...

L0  :  add  $s0, $s3, $s4  # f = i + j
     j     Exit
L1  :  add  $s0, $s1, $s2  # f = g + h
     j     Exit
L2  :  sub  $s0, $s1, $s2  # f = g – h
     j     Exit
L3  :  sub  $s0, $s1, $s2  # f = i – j
Exit :
Complex Arithmetic Example

\[ z = (a \times b) + \left( \frac{c}{d} \right) - (e + f \times g); \]

```
lw $s0,a
lw $s1,b
mult $s0,$s1
mflo $t0
lw $s0,c
lw $s1,d
div $s0,$s1
mflo $t1
add $t0,$t0,$t1
lw $s0,e
lw $s1,f
lw $s2,g
mult $s1,$s2
mflo $t1
add $t1,$s0,$t1
sub $t0,$t0,$t1
sw $t0,z
```
If-Statement

```c
if ((a>b)&&(c==d)) e=0; else e=f;
```

```assembly
lw $s0,a
lw $s1,b
bgt $s0,$s1,next0
b nope

next0: lw $s0,c
       lw $s1,d
       beq $s0,$s1,yup

nope: lw $s0,f
       sw $s0,e
       b out

yup: xor $s0,$s0,$s0
     sw $s0,e

out: ...
```
For Loop

```c
for (i=0; i<a; i++) b[i]=i;

lw $s0,a
li $s1,0

loop0: blt $s1,$s0,loop1
    b out

loop1: sll $s2,S1,2
    sw $s1,b($s2)
    addi $s1,$s1,1
    b loop0

out:      ...
```
Pre-Test While Loop

```c
while (a<b) {
    a++;
}
```

```assembly
lw $s0,a
lw $s1,b
loop0: blt $s0,$s1,loop1
        b out
loop1:  addi $s0,$s0,1
        sw $s0,a
        b loop0
out:    ...
Post-Test While Loop

do {
    a++;
} while (a<b);

lw $s0,a
lw $s1,b
loop0: addi $s0,$s0,1
sw $s0,a
blt $s0,$s1,loop0
...

for (i=0; i<n; i++) a[i] = b[i] + 10;

li $2,$0 # zero out index register (i)
lw $3,n # load iteration limit
sll $3,$3,2 # multiply by 4 (words)
la $4,a # get address of a (assume < 2^{16})
la $5,b # get address of b (assume < 2^{16})
j test
loop: add $6,$5,$2 # compute address of b[i]
lw $7,0($6) # load b[i]
addi $7,$7,10 # compute b[i] = b[i] + 10
add $6,$4,$2 # compute address of a[i]
sw $7,0($6) # store into a[i]
addi $2,$2,4 # increment i
test: blt $2,$3,loop # loop if test succeeds