Review

• Arrays, Structs, and Pointers allow you define sophisticated data structures
  – Compiler protects you by enforcing type system
  – Avoid dropping beneath the abstraction and munging the bits
• All map into untyped storage, ints, and addresses
• Executing program has a specific structure
  – Code, Static Data, Stack, and Heap
  – Mapped into address space
  – “Holes” allow stack and heap to grow
  – Compiler defines what the bits mean by enforcing type
    » Chooses which operations to perform
• Poor coding practices, bugs, and architecture limitations lead to vulnerabilities
int main() {
    ...
}

.data
A:.word 5
...
.text
main: lw $a0, x
jal decre
move $a0,$v0
Elements of the Language

• Basic Data Types: char, int, float double
• Type Constructors: array, struct, pointer
• Variables
• Expressions
• Sequence of statements
• Conditionals
• Iteration
• Functions
What does the machine do?

Instruction Execution Cycle

- Instruction Fetch
- Decode
- Operand Fetch
- Execute
- Result Store
- Next Instruction
Instruction Cycle

Instruction Fetch
Decode
Operand
Execute
Result
Next

PC 0B24

main: 0B20:
FFF.F:
n:
000..0:

"add $1,$2,$3"
Instruction Cycle - again

Instruction Fetch
Decode
Operand
Execute
Result
Next

PC

0B28

main: 0B20:

000..0:

n:

Instruction "lw $2,$1,00"

“lw $2,$1,00”
What does the machine do?

Instruction Execution Cycle

- Instruction Fetch
- Decode
- Operand Fetch
- Execute
- Result Store
- Next Instruction

Register Transfers

\[
\begin{align*}
\text{inst} & \leftarrow \text{mem}[ \text{PC} ] \\
\text{op, rd, rs, rt} & \leftarrow \text{inst} \\
A & \leftarrow \text{reg}[ \text{rs} ] \\
B & \leftarrow \text{reg}[ \text{rt} ] \\
R & \leftarrow A + B \\
\text{reg}[ \text{rd} ] & \leftarrow A + B \\
\text{PC} & \leftarrow \text{PC} + 4
\end{align*}
\]
MIPS Assembly Language (MAL)

```assembly
.data
A:    .word 5
     .word 6
     ...
.text
main: la $t0, A
     lw $a0, 4($t0)
     jal decr
     move $a0,$v0
     ...
     decr: addi $v0, $a0, -1
     jr $ra
```
MIPS Instruction Format

```
op 6   rs 5   rt 5   rd 5   shamt 5   funct 6

immediate 16
```

```
add $1, $2, $3
# r1 := r2 + r3

lw $3, 24($2)
# r3 := mem[ r2 + 24 ]
```
### MIPS register conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Use</th>
<th>Callee must preserve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>constant 0</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>assembler temporary</td>
<td>No</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>$2–$3</td>
<td>returns values</td>
<td>No</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>$4–$7</td>
<td>function arguments</td>
<td>No</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>$8–$15</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>$16–$23</td>
<td>saved temporaries</td>
<td>Yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>$24–$25</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>$26–$27</td>
<td>reserved for OS kernel</td>
<td>No</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>return address</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Administration

• Calendar with links on the home page
• Readers will be in lab 1 hour per week
• HW3R – Resubmit as hw3r permitted till 11:59 pm Saturday
  – as you learn, don’t be afraid to make a fresh start
• HW4 out – all future HW will be W-W
  – less work than hw3, little reliance on Th/F Lab, start right away
  – Continues C concepts plus basic MAL
• Project 1 posted
• Mid Term 1 shifted to Wed 10/7 in class time
  – alternative Monday 10/5 @ 4 pm
Elements of the Language

- Basic Data Types: char, int, float double
- Type Constructors: array, struct, pointer
- Variables
  - Expressions
  - Sequence of statements
  - Conditionals
  - Iteration
  - Functions
Expressions

\[
y_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a};
\]
\[
y_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a};
\]

\[
t_1 = -b;
\]
\[
t_2 = b^2;
\]
\[
t_3 = 4a;
\]
\[
t_4 = t_3c;
\]
\[
t_5 = t_2 - t_4;
\]
\[
t_6 = \sqrt{t_5};
\]
\[
t_7 = t_1 + t_6;
\]
\[
t_8 = 2a;
\]
\[
y_1 = \frac{t_7}{t_8};
\]

\[
t_1 = -b;
\]
\[
t_2 = b^2;
\]
\[
t_3 = 4a;
\]
\[
t_3 = t_3c;
\]
\[
a_0 = t_2 - t_3;
\]
\[
v_0 = \sqrt{a_0};
\]
\[
t_7 = t_1 + v_0;
\]
\[
t_8 = 2a;
\]
\[
y_1 = \frac{t_7}{t_8};
\]

\[
lw\ t_0, b
\]
\[
sub\ t_1, 0, t_0
\]
\[
add\ t_7, t_1, v_0
\]
\[
sub\ t9, t_1, v_0
\]
\[
div\ t0, t7, t8;
\]
\[
sw\ t0, y1
\]
\[
div\ t0, t9, t8;
\]
\[
sw\ t0, y2
\]
\[ y_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}; \]
\[ y_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}; \]

t1 = -b;
=> lw t0, b
=> sub t1, 0, t0

t2 = b*b;
=> mult t2, t0, t0

t3 = 4*a;
=> lw t0, a
=> mult t3, t0, 4

t3 = t3*c;
=> lw t4, c
=> mult t3, t3, t4

a0 = t2 - t3;
=> sub a0, t2, t3

v0 = sqrt(a0);
=> jal sqrt

\[ t7 = t1 + v0; \]
=> add t7, t1, v0

\[ t8 = 2*a; \]
=> mult t8, t0, 2

\[ y_1 = t7 / t8; \]
=> div t0, t7, t8;
=> sw t0, y1

\[ t9 = t1 - v0; \]
=> sub t9, t1, v0

\[ y_2 = t9 / t8; \]
=> div t0, t9, t8
=> sw t0, y2
Variables

• Can be held in Registers
  – Temporary variables
  – Internal to expression evaluation
  – Local to a function and no references to them
  – Arguments and return values

• Or in memory
  – Global or static variables (externals)
  – Local variables on the stack
  – Values in the heap

• Memory is usually accessed indirectly through a (special) register
  – stack pointer
  – global pointer
  – heap pointer
int ext;

int foo (int n) {
    int loc;
    int A [8];
    struct {int x;
            int y;} point;
    int dyn[] = malloc(10*sizeof(int));
    ...
    return (loc+n);
}
Conditionals

Human C code

```c
if (condition) {true-clause }
else {false_clause }
```

Machine-level C code

```c
if (condition) goto Ltrue;
false_clause
go to Ldone;
Ltrue:
true_clause
Ldone:
```

Machine-level Assembly code

```assembly
BR_condition Ltrue
code for false_clause
jmp Ldone
Ltrue:
code for true_clause
Ldone:
```
Jumps and Branches

• Jumps – unconditional control transfers
  – direct or indirect
  – “calls” are a special Jump-and-link
    » saves the return address
  – computes target address and loads PC

• Branches – conditional control transfers
  – tests a condition and branches if true
  – otherwise falls through sequentially
  – MIPS provides simple conditions on registers
    » BEQ, BNE, BGZ, …
Loops

Human C code

```
while (condition) {loop body }
```

Machine-level C code

```
Ltop:
if (!condition) goto Ldone;
loop body
goto Ltop;
Ldone:
```

Machine-level Assembly code

```
Ltop:
BR_condition Ltrue
jmp Ldone
Ltrue:
code for loop_body
jmp Ltop
Ldone:
```
Functions (basic)

Human C code

```c
int foo (int arg1, int arg2) {
    declarations;
    function_body;
    return val;
}

... 
res = foo(param1, param2);
... 
```

Machine-level Assembly code

```assembly
foo:
    access arg1 as $a0
    access arg2 as $a1
...
    result in $v0
    jr $ra

$a0 <= param1
$a1 <= param2
jal foo
access $v0
```
# MIPS register conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Use</th>
<th>Callee must preserve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>$0</td>
<td>constant 0</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>$1</td>
<td>assembler temporary</td>
<td>No</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>$2–$3</td>
<td>returns values</td>
<td>No</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>$4–$7</td>
<td>function arguments</td>
<td>No</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>$8–$15</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>$16–$23</td>
<td>saved temporaries</td>
<td>Yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>$24–$25</td>
<td>temporaries</td>
<td>No</td>
</tr>
<tr>
<td>$k0–$k1</td>
<td>$26–$27</td>
<td>reserved for OS kernel</td>
<td>No</td>
</tr>
<tr>
<td>$gp</td>
<td>$28</td>
<td>global pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$sp</td>
<td>$29</td>
<td>stack pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$fp</td>
<td>$30</td>
<td>frame pointer</td>
<td>Yes</td>
</tr>
<tr>
<td>$ra</td>
<td>$31</td>
<td>return address</td>
<td>N/A</td>
</tr>
</tbody>
</table>
but...

- What if the procedure calls another procedure?
  - $ra will get clobbered by the nested call!
  - How do I save it?
- The function is suppose to save any $s registers it uses. How does it do that?
- The function gets to clobber the $t registers, if the caller is still using some, how does it save them?
- What about the stack pointer?

- On the stack!
  - and a little bit of moving things between registers
The stack

Calling Conventions:
• first 4 args in $a0-$a4
• results in $v0-$v1

Hardware:
• $ra set by jal to following instruction
Functions ($ra)

main:
...
    jal foo
...
foo:
...
    jr $ra
Functions ($sp)

- Adjust the stack ptr by a constant large enough to hold the frame
- Restore before returning

```assembly
main:
...
jal foo
...
foo:
  subu $sp, $sp, 32
  ...
  addu $sp, $sp, 32
  jr $ra
```

0000:

```
$sp =>

FFFF:
```

arg 5
arg 6

caller's locals

0000:

```
$sp =>
```

9/16/09
UCB CS61CL F09 Lec 4
Functions ($ra)

• Save return address in the new frame
• Restore it before returning

```
main:
    ...
    jal foo
    ...

foo:
    subu $sp, $sp, 4  
    sw $ra, 0($sp)   
    ...
    lw $ra, 0($sp)   
    addu $sp, $sp, 4
    jr $ra
```

0000:

```
FFFF:
```
Functions ($s$ registers)

main:
...
jal foo
...

foo:
subu $sp, $sp, 16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)
...
lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addu $sp, $sp, 16
jr $ra
Functions (locals)

main:
...
    jal foo
...

foo:
subu $sp, $sp, 40
sw $s0, 24($sp)
sw $s1, 28($sp)
sw $s2, 32($sp)
sw $ra, 36($sp)
...
sw $xx, (0)$sp
...
lw $s0, 24($sp)
lw $s1, 28($sp)
lw $s2, 32($sp)
lw $ra, 36($sp)
addu $sp, $sp, 40
jr $ra

0000:

sp =>
  p
  A[0]
  A[1]
  n

sp =>
  s0
  s1
  s2
  ra

caller’s locals

FFFF:
Functions ($t$’s)

- only save $ts$ that are “live” at point of call
- anticipate need by reserving space in frame on entry

```assembly
main:
    subu $sp, $sp, 40
    ...
    sw $t1, 16($sp)
    sw $t3, 20($sp)
    jal foo
    lw $t1, 16($sp)
    lw $t3, 20($sp)
    ...
foo:
    ...
    jr $ra
```

```assembly
0000:
$sp =>
arg 5
arg 6
caller’s saved $ts
caller’s locals
FFFF:
```
Summary

• Compiler “expands” language elements into machine operations
  – declarations, sequence, conditional, iteration, function

• Register usage established “by convention”
  – hardware dictates some specific usage ($ra)
  – $sp, $gp, $a0-3, $v0-1, $s0-7, $t0-9

• Calling Convention used systematically
  – stack frame per call
  – save/restore $sp (arithmetically)
  – >4 args “pushed” before call
  – caller saves $t’s that it still wants, callee can trash any $t’s
  – callee saves $s’s that it uses, caller can assume all restored
  – $ra saved/restored if not a leaf procedure
  – Locals on the stack, discarded when $sp restored

• Enables nesting and recursion