C: Assembly Language (MIPS)

Instruction Set Architectures

CPU Architecture

A typical modern CPU has

- a set of data registers
- a set of control registers (incl PC)
- an arithmetic-logic unit (ALU)
- access to random access memory (RAM)
- a set of simple instructions
  - transfer data between memory and registers
  - push values through the ALU to compute results
  - make tests and transfer control of execution

Different types of processors have different configurations of the above

- e.g. different # registers, different sized registers, different instructions

---

Why Study Assembler?

Useful to know assembly language because ...

- sometimes you are *required* to use it (e.g. device handlers)
- improves your understanding of how C programs execute
  - very helpful when debugging
  - able to avoid using known inefficient constructs
- uber-nerdy performance tweaking (squeezing out last nano-s)
  - re-write that critical (frequently-used) function in assembler

---

Instruction Sets

Two broad families of instruction set architectures ...

RISC (*reduced instruction set computer*)

- small(ish) set of simple, general instructions
- separate computation & data transfer instructions
- leading to simpler processor hardware
- e.g. MIPS, RISC, Alpha, SPARC, PowerPC, ARM, ...

CISC (*complex instruction set computer*)

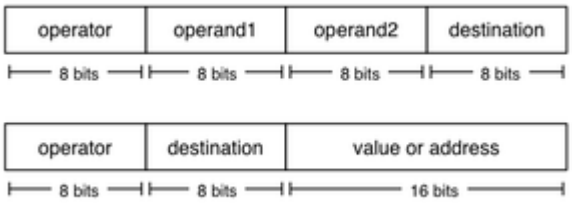
- large(r) set of powerful instructions
- each instruction has multiple actions (comp+store)
- more circuitry to decode/process instructions
- e.g. PDP, VAX, Z80, Motorola 68xxx, Intel x86, ...

... Instruction Sets

Machine-level instructions ...

- typically have 1-2 32-bit words per instruction
- partition bits in each word into operator & operands
- #bits for each depends on #instructions, #registers, ...

Example instruction word formats:



Operands and *destination* are typically registers

... Instruction Sets

6/114

Common kinds of instructions (not from any real machine)

- `load` *Register*, *MemoryAddress*
  - copy value stored in memory at address into named register
- `loadc` *Register*, *ConstantValue*
  - copy value into named register
- `store` *Register*, *MemoryAddress*
  - copy value stored in named register into memory at address
- `jump` *MemoryAddress*
  - transfer execution of program to instruction at address
- `jumpif` *Register*, *MemoryAddress*
  - transfer execution of program if e.g. register holds zero value

---

... Instruction Sets

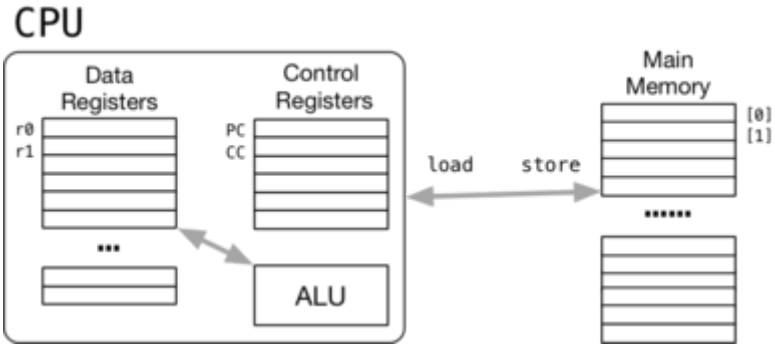
7/114

Other common kinds of instructions (not from any real machine)

- `add` $Register_1$, $Register_2$, $Register_3$ (similarly for `sub`, `mul`, `div`)
  - $Register_3 = Register_1 + Register_2$
- `and` $Register_1$, $Register_2$, $Register_3$ (similarly for `or`, `xor`)
  - $Register_3 = Register_1 \,\&\, Register_2$
- `neg` $Register_1$, $Register_2$
  - $Register_2 = \sim Register_1$
- `shiftl` $Register_1$, *Value*, $Register_2$ (similarly for `shiftr`)
  - $Register_2 = Register_1 << Value$
- `syscall` *Value*
  - invoke a system service; which service determined by *Value*

... Instruction Sets

8/114



Fetch-Execute Cycle

9/114

All CPUs have program execution logic like:

```
while (1)
{
   instruction = memory[PC]
   PC++  // move to next instr
   if (instruction == HALT)
      break
   else
      execute(instruction)
}
```

PC = Program Counter, a CPU register which keeps track of execution

Note that some instructions may modify PC further (e.g. JUMP)

... Fetch-Execute Cycle

10/114

Executing an `instruction` involves

- determine what the operator is
- determine which registers, if any, are involved
- determine which memory location, if any, is involved
- carry out the operation with the relevant operands
- store result, if any, in appropriate register

Example instruction encodings (not from a real machine):

Assembly Language

11/114

Instructions are simply bit patterns within a 32-bit bit-string

Could describe machine programs as a sequence of hex digits, e.g.

```
Address    Content
0x100000   0x3c041001
0x100004   0x34020004
0x100008   0x0000000c
0x10000C   0x03e00008
```

Often call "assembly language" as "assembler"

Slight notational abuse, because "assembler" also refers to a program that translates assembly language to machine code

---

... Assembly Language

12/114

Assembler = symbolic language for writing machine code

- write instructions using mnemonics rather than hex codes
- reference registers using either numbers or names
- can associate names to memory addresses

Style of expression is significantly different to e.g. C

- need to use fine-grained control of memory usage
- required to manipulate data in registers
- control structures programmed via explicit jumps

MIPS Architecture

13/114

MIPS is a well-known and relatively simple architecture

- very popular in a range of computing devices in the 1990's
- e.g. Silicon Graphics, NEC, Nintendo64, Playstation, supercomputers

We consider the MIPS32 version of the MIPS family

- using two variants of the open-source SPIM emulator
- `qtspim` ... provides a GUI front-end, useful for debugging
- `spim` ... command-line based version, useful for testing
- `xspim` ... GUI front-end, useful for debugging, only in CSE labs

Executables and source: http://spimsimulator.sourceforge.net/

Source code for browsing under /home/cs1521/spim/spim

MIPS vs SPIM

14/114

MIPS is a machine architecture, including instruction set

SPIM is an *emulator* for the MIPS instruction set

- reads text files containing instruction + directives
- converts to machine code and loads into "memory"
- provides debugging capabilities
  - single-step, breakpoints, view registers/memory, ...
- provides mechanism to interact with operating system (syscall)

Also provides extra instructions, mapped to MIPS core set

- provide convenient/mnemonic ways to do common operations
- e.g. `move $s0,$v0` rather than `addu $s0,$0,$v0`

Using SPIM

15/114

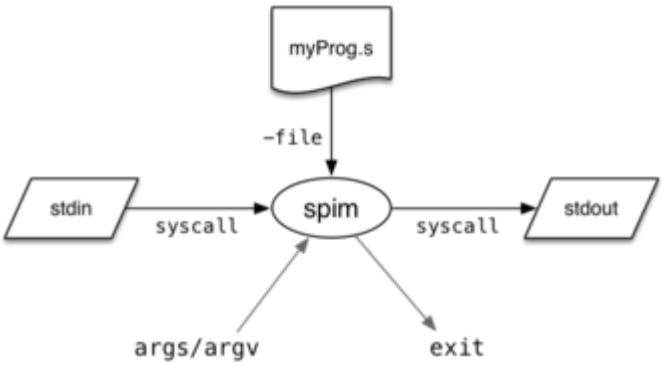
Three ways to execute MIPS code with SPIM

- `spim` ... command line tool
  - load programs using `-file` option
  - interact using stdin/stdout via login terminal
- `qtspim` ... GUI environment
  - load programs via a load button
  - interact via a pop-up stdin/stdout terminal
- `xspim` ... GUI environment
  - similar to `qtspim`, but not as pretty
  - requires X-windows server

... Using SPIM

16/114

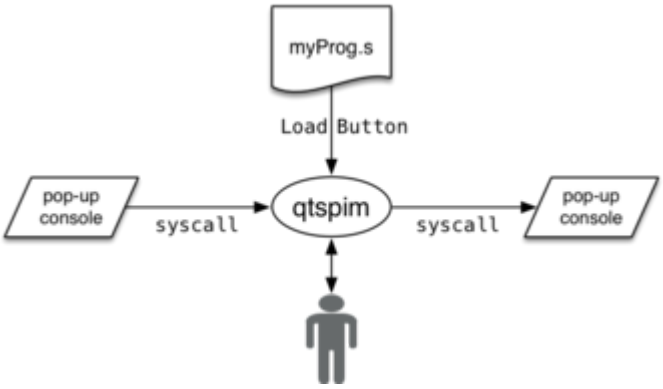
Command-line tool:



... Using SPIM

17/114

GUI tool:



MIPS Machine Archtecture

18/114

MIPS CPU has

- 32 × 32-bit general purpose registers
- 16 × 64-bit double-precision registers
- PC ... 32-bit register (always aligned on 4-byte boundary)
- HI,LO ... for storing results of multiplication and division

Registers can be referred to as `$0..$31` or by symbolic names

Some registers have special uses e.g.

- register `$0` always has value 0, cannot be written
- registers `$1`, `$26`, `$27` reserved for use by system

More details on following slides ...

... MIPS Machine Archtecture

Registers and their usage

Reg	Name	Notes
$0	zero	the value 0, not changeable
$1	$at	**a**ssembler **t**emporary; used to implement pseudo-ops
$2	$v0	**v**alue from expression evaluation or function return
$3	$v1	**v**alue from expression evaluation or function return
$4	$a0	first **a**rgument to a function/subroutine, if needed
$5	$a1	second **a**rgument to a function/subroutine, if needed
$6	$a2	third **a**rgument to a function/subroutine, if needed
$7	$a3	fourth **a**rgument to a function/subroutine, if needed
$8..$15	$t0..$t7	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite

---

... MIPS Machine Archtecture

More register usage ...

Reg	Name	Notes
$16..$23	$s0..$s7	**s**afe function variable; must not be overwritten by called subroutine
$24..$25	$t8..$t9	**t**emporary; must be saved by caller to subroutine; subroutine can overwrite
$26..$27	$k0..$k1	for **k**ernel use; may change unexpectedly
$28	$gp	**g**lobal **p**ointer
$29	$sp	**s**tack **p**ointer
$30	$fp	**f**rame **p**ointer
$31	$ra	**r**eturn **a**ddress of most recent caller

... MIPS Machine Archtecture

Floating point register usage ...

Reg	Notes
$f0..$f2	hold floating-point function results
$f4..$f10	temporary registers; not preserved across function calls
$f12..$f14	used for first two double-precision function arguments
$f16..$f18	temporary registers; used for expression evaluation
$f20..$f30	saved registers; value is preserved across function calls

Notes:

- registers come in pairs of 2 × 32-bits
- only even registers are addressed for double-precision

MIPS Assembly Language

MIPS assembly language programs contain

- comments ... introduced by #
- labels ... appended with :
- directives ... symbol beginning with .
- assembly language instructions

Programmers need to specify

- data objects that live in the data region
- functions (instruction sequences) that live in the code/text region

Each instruction or directive appears on its own line

... MIPS Assembly Language

Example MIPS assembler program:

```
# hello.s ... print "Hello, MIPS"

        .data           # the data segment
msg:    .asciiz "Hello, MIPSWn"

        .text           # the code segment
        .globl main
main:
         la $a0, msg    # load the argument string
         li $v0, 4      # load the system call (print)
         syscall        # print the string
         jr $ra         # return to caller (__start)
```

Color coding: label, directive, comment

... MIPS Assembly Language

Generic structure of MIPS programs

```
# Prog.s ... comment giving description of function
# Author ...

     .data     # variable declarations follow this line
               # ...

     .text     # instructions follow this line
     .globl main
main:          # indicates start of code
               # (i.e. first user instruction to execute)
               # ...

# End of program; leave a blank line to make SPIM happy
```

... MIPS Assembly Language

Another example MIPS assembler program:

```
    .data
a:  .word 42            # int a = 42;
b:  .space 4            # int b;
    .text
    .globl main
main:
    lw   $t0, a         # reg[t0] = a
    li   $t1, 8         # reg[t1] = 8
    add  $t0, $t0, $t1  # reg[t0] = reg[t0]+reg[t1]
    li   $t2, 666       # reg[t2] = 666
    mult $t0, $t2       # (Lo,Hi) = reg[t0]*reg[t2]
    mflo $t0            # reg[t0] = Lo
    sw   $t0, b         # b = reg[t0]
    ....
```

... MIPS Assembly Language

26/114

MIPS programs assume the following memory layout

Region	Address	Notes
text	0x00400000	contains only instructions; read-only; cannot expand
data	0x10000000	data objects; readable/writeable; can be expanded
stack	0x7fffefff	grows down from that address; readable/writeable
k_text	0x80000000	kernel code; read-only; only accessible kernel mode
k_data	0x90000000	kernel data; read/write; only accessible kernel mode

---

MIPS Instructions

27/114

MIPS has several classes of instructions:

- *load and store* .. transfer data between registers and memory
- *computational* ... perform arithmetic/logical operations
- *jump and branch* ... transfer control of program execution
- *coprocessor* ... standard interface to various co-processors
- *special* ... miscellaneous tasks (e.g. syscall)

And several addressing modes for each instruction

- between memory and register (direct, indirect)
- constant to register (immediate)
- register + register + destination register

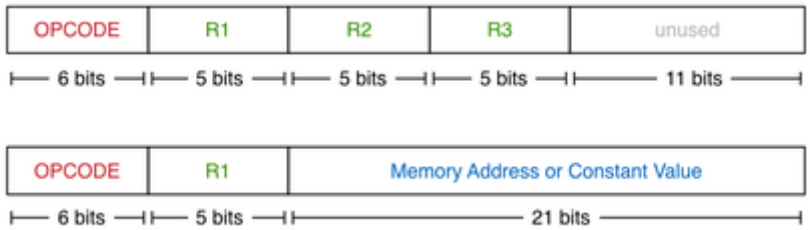
... MIPS Instructions

28/114

MIPS instructions are 32-bits long, and specify ...

- an operation (e.g. load, store, add, branch, ...)
- one or more operands (e.g. registers, memory addresses, constants)

Some possible instruction formats



Addressing Modes

29/114

Memory addresses can be given by

- symbolic name (label) (effectively, a constant)
- indirectly via a register (effectively, pointer dereferencing)

Examples:

```
prog:
a:    lw    $t0, var     # address via name
b:    lw    $t0, ($s0)   # indirect addressing
c:    lw    $t0, 4($s0)  # indexed addressing
```

If `$s0` contains `0x10000000` and `&var = 0x100000008`

- computed address for `a:` is `0x100000008`
- computed address for `b:` is `0x100000000`
- computed address for `c:` is `0x100000004`

... Addressing Modes

30/114

Addressing modes in MIPS

Format	Address computation
(register)	address = *register = contents of register
k	address = k
k(register)	address = k + *register
symbol	address = &symbol = address of symbol
symbol ± k	address = &symbol ± k
symbol ± k(register)	address = &symbol ± (k + *register)

where *k* is a literal constant value (e.g. 4 or `0x10000000`)

... Addressing Modes

31/114

Examples of load/store and addressing:

```
    .data
vec: .space   16      # int vec[4], 16 bytes of storage
    .text
__start:
    la  $t0, vec      # reg[t0] = &vec
```

```
    li  $t1, 5        # reg[t1] = 5
    sw  $t1, ($t0)    # vec[0] = reg[t1]
    li  $t1, 13       # reg[t1] = 13
    sw  $t1, 4($t0)   # vec[1] = reg[t1]
    li  $t1, -7       # reg[t1] = -7
    sw  $t1, 8($t0)   # vec[2] = reg[t1]
    li  $t2, 12       # reg[t2] = 12
    li  $t1, 42       # reg[t1] = 42
    sw  $t1, vec($t2) # vec[3] = reg[t1]
```

Operand Sizes

MIPS instructions can manipulate different-sized operands

- single bytes, two bytes ("halfword"), four bytes ("word")

Many instructions also have variants for signed and unsigned

Leads to many opcodes for a (conceptually) single operation, e.g.

- `LB` ... load one byte from specified address
- `LBU` ... load unsigned byte from specified address
- `LH` ... load two bytes from specified address
- `LHU` ... load unsigned 2-bytes from specified address
- `LW` ... load four bytes (one word) from specified address
- `LA` ... load the specified address

All of the above specify a destination register

MIPS Instruction Set

The MIPS processor implements a base set of instructions, e.g.

- `lw`, `sw`, `add`, `sub`, `and`, `or`, `sll`, `slt`, `beq`, `j`, `jal`, ...

Augmented by a set of pseudo-instructions, e.g.

- `move`, `rem`, `la`, `li`, `blt`, ...

Each pseudo-instruction maps to one or more base instructions, e.g.

```
Pseudo-instruction      Base instruction(s)

li   $t5, const         ori  $t5, $0, const

la   $t3, label         lui  $at, &label[31..16]
                        ori  $t3, $at, &label[15..0]

bge  $t1, $t2, label    slt  $at, $t1, $t2
                        beq  $at, $0, label

blt  $t1, $t2, label    slt  $at, $t1, $t2
                        bne  $at, $0, label
```

Note: use of `$at` register for intermediate results

... MIPS Instruction Set

In describing instructions:

Syntax **Semantics**

$Reg	as source, the content of the register, `reg[Reg]`
$Reg	as destination, value is stored in register, `reg[Reg] = value`
Label	references the associated address (in C terms, `&Label`)
Addr	any expression that yields an address (e.g. `Label($Reg)`)
Addr	as source, the content of memory cell `memory[Addr]`
Addr	as destination, value is stored in `memory[Addr] = value`

Effectively ...

- treat registers as `unsigned int reg[32]`
- treat memory as `unsigned char mem[`2^{32}`]`

... MIPS Instruction Set

35/114

Examples of data movement instructions:

```
la   $t1,label    # reg[t1] = &label
lw   $t1,label    # reg[t1] = memory[&label]
sw   $t3,label    # memory[&label] = reg[t3]
                  # &label must be 4-byte aligned
lb   $t2,label    # reg[t2] = memory[&label]
sb   $t4,label    # memory[&label] = reg[t4]
move $t2,$t3      # reg[t2] = reg[t3]
lui  $t2,const    # reg[t2][31:16] = const
```

Examples of bit manipulation instructions:

```
and  $t0,$t1,$t2  # reg[t0] = reg[t1] & reg[t2]
and  $t0,$t1,Imm  # reg[t0] = reg[t1] & Imm[t2]
                  # Imm is a constant (immediate)
or   $t0,$t1,$t2  # reg[t0] = reg[t1] | reg[t2]
xor  $t0,$t1,$t2  # reg[t0] = reg[t1] ^ reg[t2]
neg  $t0,$t1      # reg[t0] = ~ reg[t1]
```

... MIPS Instruction Set

36/114

Examples of arithmetic instructions:

```
add  $t0,$t1,$t2  # reg[t0] = reg[t1] + reg[t2]
                  #   add as signed (2's complement) ints
sub  $t2,$t3,$t4  # reg[t2] = reg[t3] + reg[t4]
addi $t2,$t3, 5   # reg[t2] = reg[t3] + 5
                  #   "add immediate" (no sub immediate)
addu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   add as unsigned integers
subu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   subtract as unsigned integers
mult $t3,$t4      # (Hi,Lo) = reg[t3] * reg[t4]
                  #   store 64-bit result in registers Hi,Lo
div  $t5,$t6      # Lo = reg[t5] / reg[t6] (integer quotient)
                  # Hi = reg[t5] % reg[t6] (remainder)
mfhi $t0          # reg[t0] = reg[Hi]
mflo $t1          # reg[t1] = reg[Lo]
                  # used to get result of MULT or DIV
```

... MIPS Instruction Set

37/114

Examples of testing and branching instructions:

```
seq  $t7,$t1,$t2   # reg[t7] = 1 if (reg[t1]==reg[t2])
                   # reg[t7] = 0 otherwise (signed)
slt  $t7,$t1,$t2   # reg[t7] = 1 if (reg[t1] < reg[t2])
                   # reg[t7] = 0 otherwise (signed)
slti $t7,$t1,Imm   # reg[t7] = 1 if (reg[t1] < Imm)
                   # reg[t7] = 0 otherwise (signed)

j    label         # PC = &label
jr   $t4           # PC = reg[t4]
beq  $t1,$t2,label # PC = &label if (reg[t1] == reg[t2])
bne  $t1,$t2,label # PC = &label if (reg[t1] != reg[t2])
bgt  $t1,$t2,label # PC = &label if (reg[t1] > reg[t2])
bltz $t2,label     # PC = &label if (reg[t2] < 0)
bnez $t3,label     # PC = &label if (reg[t3] != 0)
```

After each branch instruction, execution continues at new PC location

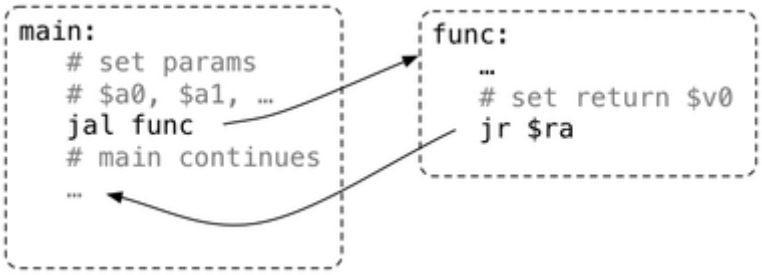
---

... MIPS Instruction Set

38/114

Special jump instruction for invoking functions

```
jal  label         # make a subroutine call
                   # save PC in $ra, set PC to &label
                   # use $a0,$a1 as params, $v0 as return
```



... MIPS Instruction Set

39/114

SPIM interacts with stdin/stdout via `syscall`s

Service	Code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = char *	
read_int	5		integer in $v0
read_float	6		float in $f0
read_double	7		double in $f0
read_string	8	$a0 = buffer, $a1 = length	string in buffer (including "₩n₩0")

... MIPS Instruction Set

40/114

Directives (instructions to assembler, not MIPS instructions)

```
    .text       # following instructions placed in text
    .data       # following objects placed in data

    .globl      # make symbol available globally

a:  .space 18   # uchar a[18];  or  uint a[4];
    .align 2    # align next object on 2^2-byte addr

i:  .word 2     # unsigned int i = 2;
v:  .word 1,3,5 # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6 # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3 # unsigned char b[3] = {1,2,3};
f:  .float 3.14 # float f = 3.14;

s:  .asciiz "abc"
                # char s[4] {'a','b','c','\0'};
t:  .ascii "abc"
                # char s[3] {'a','b','c'};
```

MIPS Programming

Writing directly in MIPS assembler is difficult (impossible?)

Strategy for producing likely correct MIPS code

- develop the solution in C
- map to "simplified" C
- translate each simplified C statement to MIPS instructions

Simplified C

- does *not* have `while`, `switch`, complex expressions
- *does* have simple `if`, `goto`, one-operator expressions
- does *not* have function calls and auto local variables
- *does* have jump-and-remember-where-you-came-from

... MIPS Programming

Example translating C to MIPS:

```
C             Simplified C   MIPS Assembler
------------  -------------  ---------------
int x = 5;    int x = 5;     x:  .word 5
int y = 3;    int y = 3;     y:  .word 3
int z;        int z;         z:  .space 4
              int t;         ...
                             lw  $t0, x
                             lw  $t1, y
z = 5*(x+y);   t = x+y;      add $t0, $t0, $t1
                             li  $t1, 5
               t = 5*t;      mul $t0, $t0, $t1
               z = t;        sw  $t0, z
```

... MIPS Programming

Simplified C makes extensive use of

- labels ... symbolic name for C statement
- goto ... transfer control to labelled statement

Example:

```
Standard C              Simplified C
-------------------     -------------------
i = 0; n = 0;           i = 0; n = 0;
while (i < 5) {         loop:
   n = n + i;             if (i >= 5) goto end;
   i++;                   n = n + i;
}                         i++;
                          goto loop;
                        end:
```

... MIPS Programming

Beware: registers are shared by all parts of the code.

One function can overwrite value set by another function

```
int x;   // first global variable
int y;   // second global variable

int main(void)          int f(int n)
{                       {
   x = 5;                  y = 1;
   y = f(x);               for (x = 1; x <= n; x++)
   printf("...",x,y);         y = y * x;
   return 0;               return y;
}                       }
```

After the function, `x == 6` and `y == 120`

It is sheer coincidence that `y` has the correct value.

... MIPS Programming

Need to be careful managing registers

- follow the conventions implied by register names
- preserve values that need to be saved across function calls

Within a function

- you manage register usage as you like
- typically making use of `$t?` registers

When making a function call

- you transfer control to a separate piece of code
- which may change the value of any non-preserved register
- `$s?` registers must be preserved by function
- `$a?`, `$v?`, `$t?` registers may be modified by function

Rendering C in MIPS

C provides expression evaluation and assignment, e.g.

- `x = (1 + y*y) / 2;  z = 1.0 / 2;` ...

MIPS provides register-register operations, e.g.

- `move` R_d,R_s, `li` R_d,Const, `add`, `div`, `and`, ...

C provides a range of control structures

- sequence (;), `if`, `while`, `for`, `break`, `continue`, ...

MIPS provides testing/branching instructions

- `seq`, `slti`, `sltu`, ..., `beq`, `bgtz`, `bgezal`, ..., `j`, `jr`, `jal`, ...

We need to render C's structures in terms of testing/branching

Sequence is easy S_1 ; S_2 → mips(S_1) mips(S_2)

... Rendering C in MIPS

47/114

Simple example of assignment and sequence:

```
int x;          x: .space 4
int y;          y: .space 4

x = 2;             li   $t0, 2
                   sw   $t0, x

y = x;             lw   $t0, x
                   sw   $t0, y

y = x+3;           lw   $t0, x
                   addi $t0, 3
                   sw   $t0, y
```

Arithmetic Expressions

48/114

Expression evaluation involves

- describing the process as a sequence of binary operations
- managing data flow between the operations

Example:

```
# x = (1 + y*y) / 2
# assume x and y exist as labels in .data
  lw   $t0, y          # t0 = y
  mul  $t0, $t0, $t0   # t0 = t0*t0
  addi $t0, $t0, 1     # t0 = t0+1
  li   $t1, 2          # t1 = 2
  div  $t0, $t1        # Lo = t0/t1 (int div)
  mflo $t0             # t0 = Lo
  sw   $t0, x          # x = t0
```

It is useful to minimise the number of registers involved in the evaluation

---

Conditional Statements

49/114

Conditional statements (e.g. `if`)

```
Standard C                Simplified C
-----------------------   -------------------------

                          if_stat:
if (Cond)                   t0 = (Cond)
   { Statements1 }          if (t0 == 0)
else                           goto else_part;
   { Statements2 }          Statements1
                            goto end_if;
                        else_part:
                           Statements2
                        end_if:
```

... Conditional Statements

Conditional statements (e.g. `if`)

```
if (Cond)                  if_stat:
   { Statements1 }           t0 = evaluate (Cond)
else                         beqz $t0, else_part
   { Statements2 }           execute Statements1
                             j    end_if
                          else_part:
                             execute Statements2
                          end_if:
```

... Conditional Statements

Example of if-then-else:

```
int x;           x is $t0
int y;           y is $t1
char z;          z is $a0

x = getInt();    li   $v0, 5
                 syscall
                 move $t0, $v0

y = getInt();    li   $v0, 5
                 syscall
                 move $t1, $v0

if (x == y)      bne  $t0, $t1, printN
   z = 'Y';    setY:
                 li   $a0, 'Y'
                 j    print
else           setN:
   z = 'N';      li   $a0, 'N'
                 j    print    # redundant
               print:
putChar(z);      li   $v0, 11
                 syscall
```

... Conditional Statements

Could make `switch` by first converting to `if`

```
switch (Expr) {              tmp = Expr;
case Val1:                   if (tmp == Val1)
   Statements1 ; break;         { Statements1; }
case Val2:                   else if (tmp == Val2
case Val3:                            || tmp == Val3
case Val4:                            || tmp == Val4)
   Statements2 ; break;         { Statements2; }
case Val5:                   else if (tmp == Val5)
   Statements3 ; break;         { Statements3; }
default:                     else
   Statements4 ; break;         { Statements4; }
}
```

... Conditional Statements

Jump table: an alternative implementation of `switch`

- works best for small, dense range of case values (e.g. 1..10)

```
                            jump_tab:
                               .word c1, c2, c2, c2, c3
                            switch:
                                t0 = evaluate Expr
switch (Expr) {                 if (t0 < 1 || t0 > 5)
case 1:                            jump to default
   Statements1 ; break;         dest = jump_tab[(t0-1)*4]
case 2:                         jump to dest
case 3:                     c1: execute Statements1
case 4:                         jump to end_switch
   Statements2 ; break;     c2: execute Statements2
case 5:                         jump to end_switch
   Statements3 ; break;     c3: execute Statements3
default:                        jump to end_switch
   Statements4 ; break;     default:
}                               execute Statements4
                            end_switch:
```

Boolean Expressions

54/114

Boolean expressions in C are short circuit

(Cond$_1$ && Cond$_2$ && ... && Cond$_n$)

Evaluates by

- evaluate Cond$_1$; if 0 then return 0 for whole expression
- evaluate Cond$_2$; if 0 then return 0 for whole expression
- ...
- evaluate Cond$_n$; if 0 then return 0 for whole expression
- otherwise, return 1

In C, any non-zero value is treated as true; MIPS tends to use 1 for true

C99 standard defines return value for booleans expressions as 0 or 1

... Boolean Expressions

55/114

Similarly for disjunctions

(Cond$_1$ || Cond$_2$ || ... || Cond$_n$)

Evaluates by

- evaluate Cond$_1$; if !0 then return 1 for whole expression
- evaluate Cond$_2$; if !0 then return 1 for whole expression
- ...
- evaluate Cond$_n$; if !0 then return 1 for whole expression
- otherwise, return 1

In C, any non-zero value is treated as true; MIPS tends to use 1 for true

C99 standard defines return value for booleans expressions as 0 or 1

Iteration Statements

56/114

Iteration (e.g. `while`)

```
                        top_while:
while (Cond) {             t0 = evaluate Cond
   Statements;             beqz $t0,end_while
}                          execute Statements
                           j    top_while
                        end_while:
```

Treat `for` as a special case of `while`

```
                                  i = 0
for (i = 0; i < N; i++) {         while (i < N) {
    Statements;                      Statements;
}                                    i++;
                                  }
```

... Iteration Statements

Example of iteration over an array:

```
int sum, i;                sum: .word 4       # use reg for i
int a[5] = {1,3,5,7,9};    a:   .word 1,3,5,7,9
...                             ...
sum = 0;                        li   $t0, 0   # i = 0
                                li   $t1, 0   # sum = 0
                                li   $t2, 4   # max index
for (i = 0; i < N; i++)    for: bgt  $t0, $t2, end_for
                                move $t3, $t0
                                mul  $t3, $t3, 4
   sum += a[i];                 add  $t1, $t1, a($t3)
printf("%d",sum);               addi $t0, $t0, 1   # i++
                                j    for
                        end_for: sw   $t1, sum
                                move $a0, $t1
                                li   $v0, 1
                                syscall       # printf
```

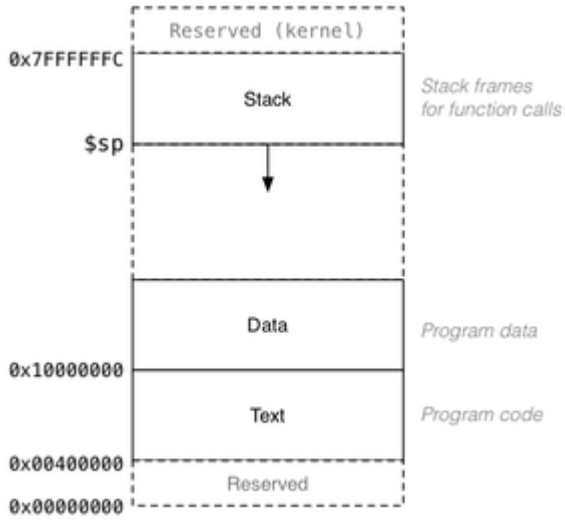
Functions

When we call a function:

- the arguments are evaluated and set up for function
- control is transferred to the code for the function
- local variables are created
- the function code is executed in this environment
- the return value is set up
- control transfers back to where the function was called from
- the caller receives the return value

... Functions

Data associated with function calls is placed on the MIPS stack.



... Functions

60/114

Each function allocates a small section of the stack (a *frame*)

- used for: saved registers, local variables, parameters to callees
- created in the function *prologue* (pushed)
- removed in the function *epilogue* (popped)

Why we use a stack:

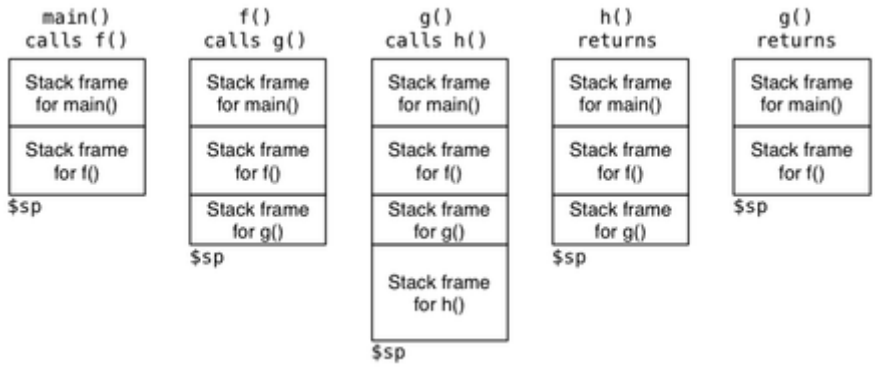
- function `f()` calls `g()` which calls `h()`
- `h()` runs, then finishes and returns to `g()`
- `g()` continues, then finishes and returns to `f()`

i.e. last-called, exits-first (last-in, first-out) behaviour

... Functions

61/114

How stack changes as functions are called and return:



... Functions

62/114

Register usage conventions when `f()` calls `g()`:

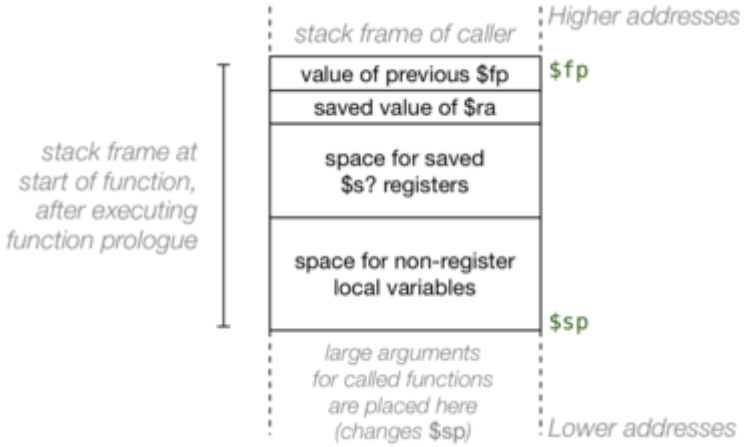
- caller saved registers (saved by `f()`)
  - `f()` tells `g()` "If there is anything I want to preserve in these registers, I have already saved it before calling you"
  - `g()` tells `f()` "Don't assume that these registers will be unchanged when I return to you"

- e.g. `$t0` .. `$t9`, `$a0` .. `$a3`, `$ra`

- callee saved registers (saved by `g()`)
  - `f()` tells `g()` "I assume the values of these registers will be unchanged when you return"
  - `g()` tells `f()` "If I need to use these registers, I will save them first and restore them before returning"
  - e.g. `$s0` .. `$s7`, `$sp`, `$fp`

... Functions

Contents of a typical stack frame:



Aside: MIPS Branch Delay Slots

The real MIPS architecture is "pipelined" to improve efficiency

- one instruction can start before the previous one finishes

For branching instructions (e.g. `jal`) ...

- instruction following branch is executed before branch completes

To avoid potential problems use `nop` immediately after branch

A problem scenario, and its solution (branch delay slot):

```
# Implementation of print(compute(42))
li   $a0, 42          li   $a0, 42
jal  compute          jal  compute
move $a0, $v0         nop
jal  print            move $a0,$v0
                      jal  print
```

Since SPIM is not pipelined, the `nop` is not required

Aside: Why do we need both $fp and $sp?

During execution of a function

- `$sp` can change (e.g. pushing params, adding local vars)
- may need to reference local vars on the stack
- useful if they can be defined by an offset relative to fixed point
- `$fp` provides a fixed point during function code execution

```
int f(int x) {
```

```
    int y = 0;                  // y created in prologue
    for (int i = 0; i < x; i++) // i created in for-loop
       y += i;                  //     which changes $sp
    return y;
}
```

Function Calling Protocol

Before one function calls another, it needs to

- place 64-bit double args in `$f12` and `$f14`
- place 32-bit arguments in the `$a0`..`$a3`
- if more than 4 args, or args larger than 32-bits ...
  - push value of all such args onto stack
- save any non-`$s?` registers that need to be preserved
  - push value of all such registers onto stack
- `jal` address of function (usually given by a label)

Pushing value of e.g. `$t0` onto stack means:

```
addi $sp, $sp, -4
sw   $t0, ($sp)
```

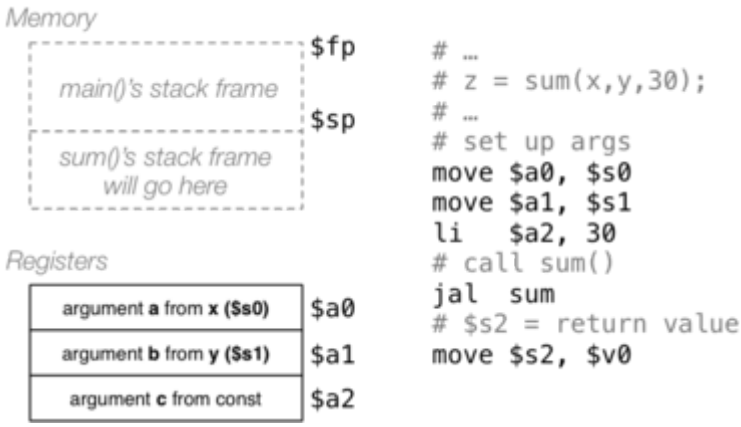
... Function Calling Protocol

Example: simple function call

```
int main()
{
   // x is $s0, y is $s1, z is $s2
   int x = 5; int y = 7; int z;
   ...
   z = sum(x,y,30);
   ...
}

int sum(int a, int b, int c)
{
   return a+b+c;
}
```

... Function Calling Protocol

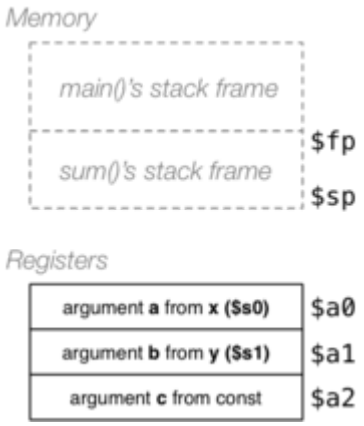
Simple function call:




```
# …
# z = sum(x,y,30);
# …
# set up args
move $a0, $s0
move $a1, $s1
li   $a2, 30
# call sum()
jal  sum
# $s2 = return value
move $s2, $v0
```

... Function Calling Protocol

Execution of `sum()` function:




```
sum:
# set up stack frame

move $v0, $a0
add  $v0, $v0, $a1
add  $v0, $v0, $a2

# remove stack frame
jr   $ra
```

... Function Calling Protocol

70/114

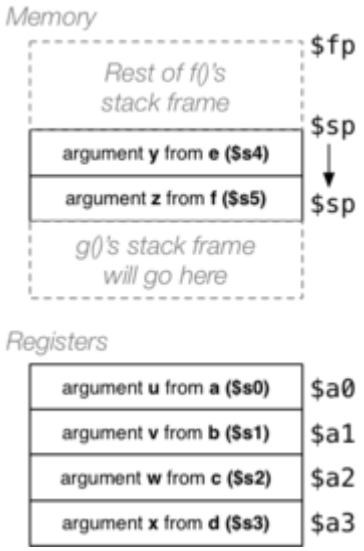
Example: function `f()` calls function `g(a,b,c,d,e,f)`

```
int f(...)
{
   // variables happen to be stored
   // in registers $s0, $s1, ..., $s5
   int a,b,c,d,e,f;
   ...
   a = g(a,b,c,d,e,f);
   ...
}
int g(int u,v,w,x,y,z)
{
   return u+v+w*w*x*y*z;
}
```

... Function Calling Protocol

71/114

Function call in MIPS:

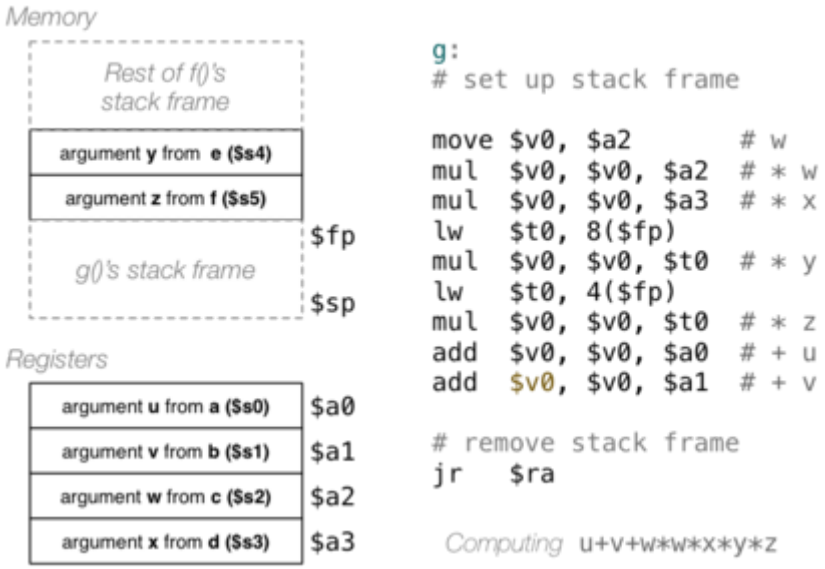



```
# … rest of f()'s code
# a = g(a,b,c,d,e,f)
# …
# set up args
move $a0, $s0
move $a1, $s1
move $a2, $s2
move $a3, $s3
sw   $s4, -4($sp)
sw   $s5, -8($sp)
addi $sp, $sp, -8
# call g()
jal  g
# $s0 = return value
move $s0, $v0
```

... Function Calling Protocol

72/114

Execution of `g()` function:



Structure of Functions

73/114

Functions in MIPS have the following general structure:

```
# start of function
FuncName:
   # function prologue
   #   set up stack frame ($fp, $sp)
   #   save relevant registers (incl. $ra)
   ...
   # function body
   #   perform computation using $a0, etc.
   #   leaving result in $v0
   ...
   # function epilogue
   #   restore saved registers (esp. $ra)
   #   clean up stack frame ($fp, $sp)
   jr  $ra
```

Aim of prologue: create environment for function to execute in.

Function Prologue

74/114

Before a function starts working, it needs to ...

- create a stack frame for itself (change `$fp` and `$sp`)
- save the return address (`$ra`) in the stack frame
- save any `$s?` registers that it plans to change

We can determine the initial size of the stack frame via

- 4 bytes for saved `$fp` + 4 bytes for saved `$ra`
- + 4 bytes for each saved `$s?`

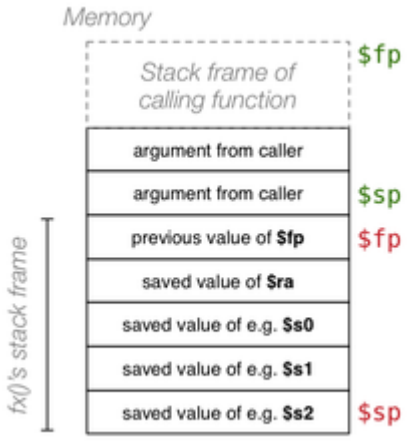
Changing `$fp` and `$sp` ...

- new `$fp` = old `$sp` - 4
- new `$sp` = old `$sp` - size of frame (in bytes)

... Function Prologue

75/114

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`

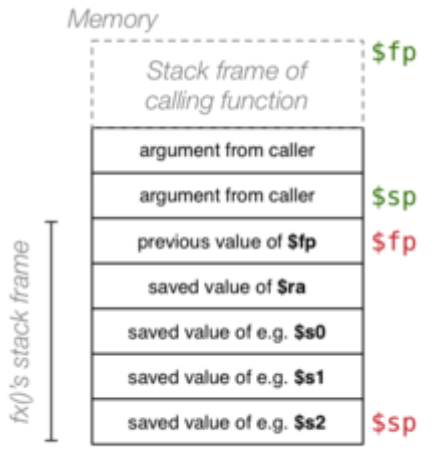



```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
sw   $ra, -8($sp)
sw   $s0, -12($sp)
sw   $s1, -16($sp)
sw   $s2, -20($sp)
la   $fp, -4($sp)
add  $sp, $sp, -20
# rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```

... Function Prologue

76/114

Alternatively ... (more explicit push)

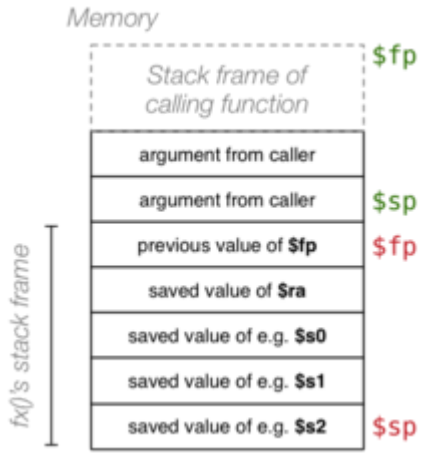



```
# start of fx() function
fx:
# fx()'s prologue
addi $sp, $sp, -4
sw   $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw   $ra, ($sp)
addi $sp, $sp, -4
sw   $s0, ($sp)
addi $sp, $sp, -4
sw   $s1, ($sp)
addi $sp, $sp, -4
sw   $s2, ($sp)
# rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```

... Function Prologue

77/114

Alternatively ... (relative to new `$fp`)




```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
la   $fp, -4($sp)
sw   $ra, -4($fp)
sw   $s0, -8($fp)
sw   $s1, -12($fp)
sw   $s2, -16($fp)
add  $sp, $sp, -20
# rest of fx()'s code

initial values: $fp,$sp
final values: $fp,$sp
```

Function Epilogue

78/114

Before a function returns, it needs to ...

- place the return value in `$v0` (and maybe `$v1`)
- pop any pushed arguments off the stack
- restore the values of any saved `$s?` registers
- restore the saved value of `$ra` (return address)
- remove its stack frame (change `$fp` and `$sp`)
- return to the calling function (`jr $ra`)

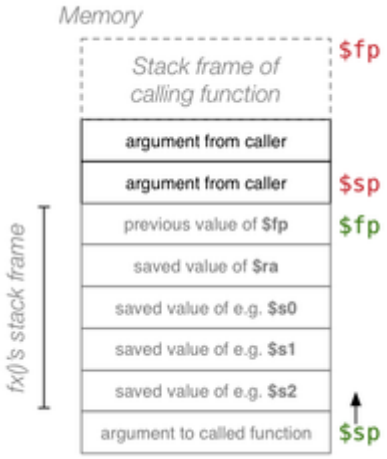
Locations of saved values computed relative to `$fp`

Changing `$fp` and `$sp` ...

- new `$sp` = old `$fp` + 4
- new `$fp` = `memory[old $fp]`

... Function Epilogue

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`




```
# end of fx() function
lw   $v0, return value
# fx()'s epilogue
lw   $s2, -16($fp)
lw   $s1, -12($fp)
lw   $s0, -8($fp)
lw   $ra, -4($fp)
# remove stack frame
la   $sp, 4($fp)
lw   $fp, ($fp)
# return
jr   $ra

initial values: $fp,$sp
final values: $fp,$sp
```

Data Structures and MIPS

C data structures and their MIPS representations:

- `char` ... as byte in memory, or low-order byte in register
- `int` ... as word in memory, or whole register
- `double` ... as two-words in memory, or `$f?` register
- arrays ... sequence of memory bytes/words, accessed by index
- structs ... chunk of memory, accessed by fields/offsets
- linked structures ... struct containing address of another struct

A `char`, `int` or `double`

- could be implemented in register if used in small scope
- could be implemented on stack if local to function
- could be implemented in `.data` if need longer persistence

Static vs Dynamic Allocation

Static allocation:

- uninitialised memory allocated at compile/assemble-time, e.g.

```
int  val;              val: .space 4
char str[20];          str: .space 20
```

```
int  vec[20];          vec: .space 80
```

- initialised memory allocated at compile/assemble-time, e.g.

```
int val = 5;               val: .word 5
int arr[4] = {9,8,7,6};    arr: .word 9, 8, 7, 6
char *msg = "HelloWn";     msg: .asciiz "HelloWn"
```

... Static vs Dynamic Allocation

82/114

Dynamic allocation (i):

- variables local to a function

Prefer to put local vars in registers, but if cannot ...

- use space allocated on stack during function prologue
- referenced during function relative to `$fp`
- space reclaimed from stack in function epilogue

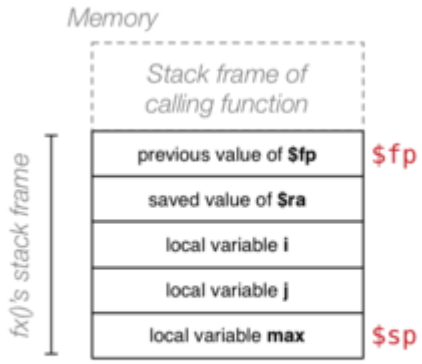
Example:

```
int fx(int a[])
{
   int i, j, max;
   i = 1; j = 2; max = i+j;
   ...
}
```

... Static vs Dynamic Allocation

83/114

Example of local variables on the stack:




```
fx:
# fx()'s prologue
# save $fp, $ra; fix $fp, $sp
# allocate space for i, j, max
add  $sp, $sp, -12
…
li   $t0, 1        # i = 1;
sw   $t0, -8($fp)
li   $t0, 2        # j = 2;
sw   $t0, -12($fp)
…
# fx()'s epilogue
add  $sp, $sp, 12
lw   $ra, ($sp)
…
```

... Static vs Dynamic Allocation

84/114

Dynamic allocation (ii):

- uninitialised block of memory allocated at run-time

```
int  *ptr = malloc(sizeof(int));
char *str = malloc(20*sizeof(char));
int  *vec = malloc(20*sizeof(int));

*ptr = 5;
strcpy(str, "a string");
vec[0] = 1;  // or  *vec = 1;
vec[1] = 6;
```

- initialised block of memory allocated at run-time

```
int *vec = calloc(20, sizeof(int));
// vec[i] == 0, for i in 0..19
```

... Static vs Dynamic Allocation

SPIM doesn't provide `malloc()`/`free()` functions

- but provides `syscall 9` to extend `.data`
- before `syscall`, set `$a0` to the number of bytes requested
- after `syscall`, `$v0` holds start address of allocated chunk

Example:

```
li   $a0, 20    # $v0 = malloc(20)
li   $v0, 9
syscall
move $s0, $v0   # $s0 = $v0
```

Cannot access allocated data by name; need to retain address.

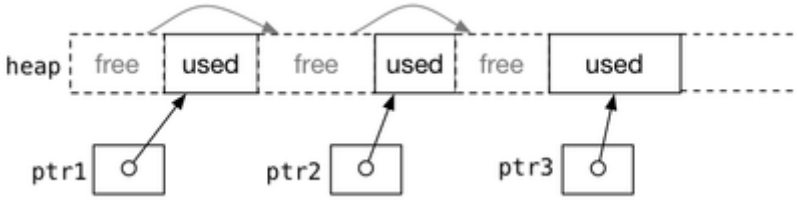
No way to free allocated data, and no way to align data appropriately

---

... Static vs Dynamic Allocation

Implementing C-like `malloc()` and `free()` in MIPS requires

- a complete implementation of C's heap management, i.e.
- a large region of memory to manage (`syscall 9`)
- ability to mark chunks of this region as "in use" (with size)
- ability to maintain list of free chunks
- ability to merge free chunks to prevent fragmentation



1-d Arrays in MIPS

Can be named/initialised as noted above:

```
vec:  .space 40
# could be either int vec[10] or char vec[40]

nums: .word 1, 3, 5, 7, 9
# int nums[6] = {1,3,5,7,9}
```

Can access elements via index or cursor (pointer)

- either approach needs to account for size of elements

Arrays passed to functions via pointer to first element

- must also pass array size, since not available elsewhere

See `sumOf()` exercise for an example of passing an array to a function

---

... 1-d Arrays in MIPS

Scanning across an array of N elements using index

```
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
#    printf("%dWn", vec[i]);

   li   $s0, 0              # i = 0
   li   $s1, 10             # no of elements
   li   $s2, 4              # sizeof each element
loop:
   bge  $s0, $s1, end_loop  # if (i >= 10) break
   mul  $t0, $s0, $s2       # index -> byte offset
   lw   $a0, vec($t0)       # a0 = vec[i]
   jal  print               # print a0
   addi $s0, $s0, 1         # i++
   j    loop
end_loop:
```

Assumes the existence of a `print()` function to do `printf("%dWn",x)`

... 1-d Arrays in MIPS

Scanning across an array of N elements using cursor

```
# int vec[10] = {...};
# int *cur, *end = &vec[10];
# for (cur = vec; cur < end; cur++)
#    printf("%dWn", *cur);

   la   $s0, vec            # cur = &vec[0]
   la   $s1, vec+40         # end = &vec[10]
loop:
   bge  $s0, $s1, end_loop  # if (cur >= end) break
   lw   $a0, ($s0)          # a0 = *cur
   jal  print               # print a0
   addi $s0, $s0, 4         # cur++
   j    loop
end_loop:
```

Assumes the existence of a `print()` function to do `printf("%dWn",x)`

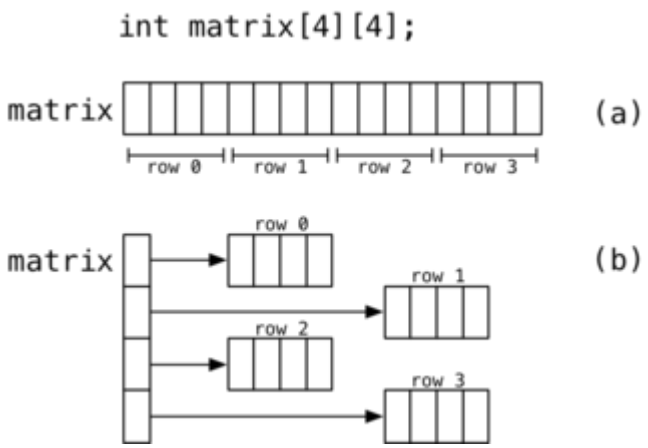
... 1-d Arrays in MIPS

Arrays that are local to functions are allocated space on the stack

```
fun:                        int fun(int x)
   # prologue               {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)             // push a[] onto stack
   addi $sp, $sp, -40          int a[10];
   move $s0, $sp               int *s0 = a;
   # function body
   ... compute ...             // compute using s0
   # epilogue                  // to access a[]
   addi $sp, $sp, 40           // pop a[] off stack
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra                 }
```

2-d Arrays in MIPS

2-d arrays could be represented two ways:



... 2-d Arrays in MIPS

Representations of `int matrix[4][4]` ...

```
# for strategy (a)
matrix: .space 64
# for strategy (b)
row0:   .space 16
row1:   .space 16
row2:   .space 16
row3:   .space 16
matrix: .word row0, row1, row2, row3
```

Now consider summing all elements

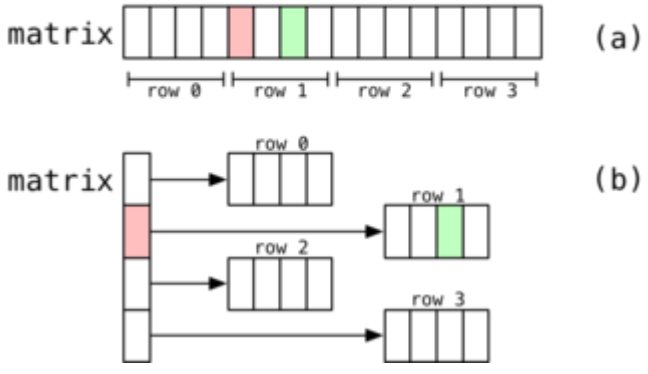
```
int i, j, sum = 0;
for (i = 0; i < 4; i++)
   for (j = 0; j < 4; j++)
      sum += matrix[i][j];
```

... 2-d Arrays in MIPS

Accessing elements:

```
x = matrix[1][2];
```

Find start of row 1, then add offset 2 within row



... 2-d Arrays in MIPS

Computing sum of all elements for strategy (a) `int matrix[4][4]`

```
    li  $s0, 0            # sum = 0
    li  $s1, 4            # s1 = 4 (and sizeof int)
    li  $s2, 0            # i = 0
    li  $s3, 16           # sizeof row in bytes
loop1:
    beq  $s2, $s1, end1   # if (i >= 4) break
    li   $s3, 0           # j = 0
loop2:
    beq  $s3, $s1, end2   # if (j >= 4) break
    mul  $t0, $s2, $s3    # off = 4*4*i + 4*j
    mul  $t1, $s3, $s1    #  matrix[i][j] is
    add  $t0, $t0, $t1    #  done as *(matrix+off)
    lw   $t0, matrix($t0) # t0 = matrix[i][j]
    add  $s0, $s0, $t0    # sum += t0
    addi $s3, $s3, 1      # j++
    j    loop2
end2:
    addi $s2, $s2, 1      # i++
    j    loop1
end1:
```

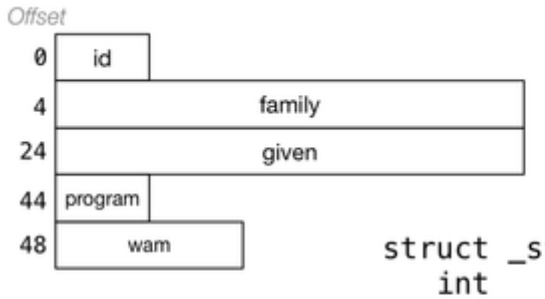
... 2-d Arrays in MIPS

Computing sum of all elements for strategy (b) `int matrix[4][4]`

```
    li  $s0, 0            # sum = 0
    li  $s1, 4            # s1 = 4 (sizeof(int))
    li  $s2, 0            # i = 0
loop1:
    beq  $s2, $s1, end1   # if (i >= 4) break
    li   $s3, 0           # j = 0
    mul  $t0, $s2, $s1    # off = 4*i
    lw   $s4, matrix($t0) # row = &matrix[i][0]
loop2:
    beq  $s3, $s1, end2   # if (j >= 4) break
    mul  $t0, $s3, $s1    # off = 4*j
    add  $t0, $t0, $s4    # int *p = &row[j]
    lw   $t0, ($t0)       # t0 = *p
    add  $s0, $s0, $t0    # sum += t0
    addi $s3, $s3, 1      # j++
    j    loop2
end2:
    addi $s2, $s2, 1      # i++
    j    loop1
end1:
```

Structs in MIPS

C `structs` hold a collection of values accessed by name




```
struct _student {
    int    id;
    char   family[20];
    char   given[20];
    int    program;
    double wam;
};
```

... Structs in MIPS

C `struct` definitions effectively define a new type.

```
// new type called struct _student
struct _student {...};
// new type called Student
typedef struct _student Student;
```

Instances of structures can be created by allocating space:

```
                // sizeof(Student) == 56
stu1:           Student stu1;
   .space 56
stu2:           Student stu2;
   .space 56
stu:
   .space 4     Student *stu;
```

... Structs in MIPS

Accessing structure components is by offset, not name

```
li  $t0  5012345
sw  $t0, stu1+0       # stu1.id = 5012345;
li  $t0, 3778
sw  $t0, stu1+44      # stu1.program = 3778;
la  $s1, stu2         # stu = & stu2;
li  $t0, 3707
sw  $t0, 44($s1)      # stu->program = 3707;
li  $t0, 5034567
sw  $t0, 0($s1)       # stu->id = 5034567;
```

... Structs in MIPS

Structs that are local to functions are allocated space on the stack

```
fun:                        int fun(int x)
   # prologue               {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)             // push onto stack
   addi $sp, $sp, -56          Student st;
   move $t0, $sp               Student *t0 = &st;
   # function body
   ... compute ...             // compute using t0
   # epilogue                  // to access struct
   addi $sp, $sp, 56           // pop st off stack
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra                 }
```

... Structs in MIPS

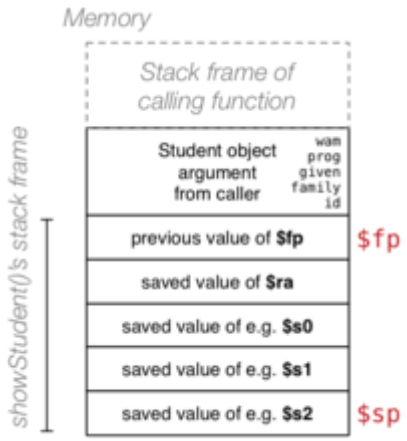
C can pass whole structures to functions, e.g.

```
# Student stu; ...
# // set values in stu struct
# showStudent(stu);
```

```
   .data
stu: .space 56
   .text
   ...
   la   $t0, stu
   addi $sp, $sp, -56    # push Student object onto stack
   lw   $t1, 0($t0)      # allocate space and copy all
   sw   $t1, 0($sp)      # values in Student object
   lw   $t1, 4($t0)      # onto stack
   sw   $t1, 4($sp)
   ...
   lw   $t1, 52($t0)     # and once whole object copied
   sw   $t1, 52($sp)
   jal  showStudent      # invoke showStudent()
   ...
```

... Structs in MIPS

Accessing `struct` within function ...




```
# start of function
showStudent:
# prologue
   …
# function body
   …
   # t0 = s.id
   lw   $t0, 4($fp)
   # t1 = s.prog
   lw   $t1, 48($fp)
   # print s.family
   la   $a0, 8($fp)
   li   $v0, 4
   syscall
   …
```

... Structs in MIPS

Can also pass a pointer to a `struct`

```
# Student stu;
# // set values in stu struct
# changeWAM(&stu, float newWAM);

   .data
stu: .space 56
wam: .space 4
   .text
   ...
   la   $a0, stu
   lw   $a1, wam
   jal  changeWAM
   ...
```

Clearly a more efficient way to pass a large `struct`

Also, required if the function needs to update the original `struct`

Compiling C to MIPS

Using simplified C as an intermediate language

- makes things easier for a human to prodcue MIPS code

- does not provide an automatic way of translating
- this is provided by a *compiler* (e.g. `dcc`)

What does the compiler need to do to convert C to MIPS?

- convert `#include` and `#define`
- *parse* code to check syntactically valid
- manage a list of *symbols* used in program
- decide how to represent data structures
- allocate local variables to registers or stack
- map control structures to MIPS instructions

C Pre-processor

104/114

Maps C→C, performing various *substitutions*

- `#include` *File*
  - replace `#include` by contents of file
  - `"name.h"` ... uses named *File*`.h`
  - <name.h> ... uses *File*`.h` in `/usr/include`
- `#define` *Name Constant*
  - replace all occurences of symbol *Name* by *Constant*
  - e.g `#define MAX 5`
    `char array[MAX]` → `char array[5]`
- `#define` *Name*`(`*Params*`)` *Expression*
  - replace *Name*`(`*Params*`)` by *SubstitutedExpression*
  - e.g. `#define max(x,y) ((x > y) ? x : y)`
    `a = max(b,c)` → `a = ((b > c) ? b : c)`

... C Pre-processor

105/114

More C pre-processor substitions

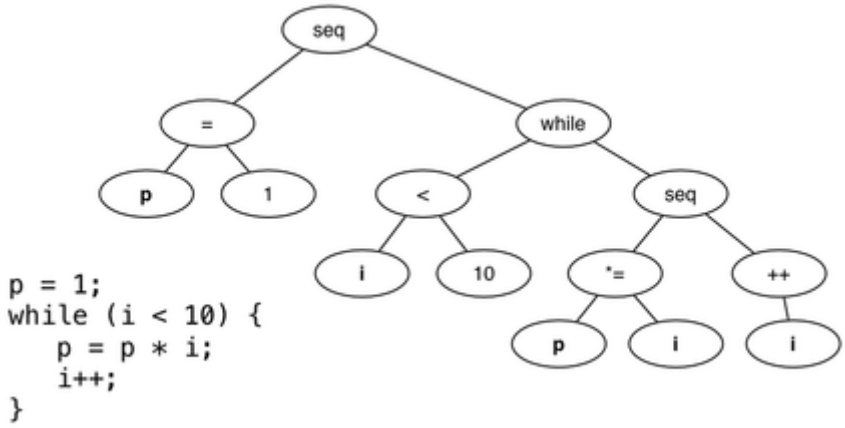
```
Before cpp                  After cpp

x = 5;                      x = 5;
#if 0                       x = x + 2;
x = x + 1;                  printf("x=%dWn",x);
#else                       x = x * 2;
x = x + 2;
#endif
                            Assuming ...
#ifdef DEBUG                #define DEBUG 1
printf("x=%dWn",x);         or
#endif                      gcc -DDEBUG=1 ...
x = x * 2;
```

C Parser

106/114

Understands syntax of C language

Attempts to convert C program into *parse tree*



Symbol Table Management

107/114

Compiler keeps track of names

- scope, lifetime, locally/externally defined
- disambiguates e.g. `x` in `main()` vs `x` in `fun()`
- resolves symbols to specific locations (data/stack/registers)
- external symbols may remain unresolved until linking
- however, need to have a type for each external symbol

Example:

```
double fun(double x, int n);

int main(void) {
   int i;  double res;
   scanf("%d", &i);
   res = fun((float)i, 5);
   return 0;
}
```

Local Variables

108/114

Two choices for local variables

- on the stack ... +persist for whole function, -`lw`/`sw` needed in MIPS
- in a register ... +efficient, -not many, useful if var used in small scope
  - if need to persist across function calls, use `$s?` register
  - if used in very localised scope, can use `$t?` register

Example:

```
int sum(List L)
{
   if (L == NULL) return 0;
   int first = L->value;     // must be in $s?
   int rest = sum(L->next);  // can be in $t?
   return first + rest;
}
```

Expression Evaluation

109/114

Uses temporary (`$t?`) registers

- even complex expressions don't generally need > 3-4 registers

Example:

```
x = ((y+3) * (z-2) * x) / 4;
```

```
   lw   $t0, y
   addi $t0, $t0, 3    # t0 = y + 3
   lw   $t1, z
   addi $t1, $t1, -2   # t1 = z - 2
   mul  $t0, $t0, $t1  # t0 = t0 * t1
   lw   $t1, x
   mul  $t0, $t0, $t1  # t0 = t0 * x
   li   $t1, 4
   div  $t0, $t0, $t1  # t0 = t0 / 4
```

Complex boolean expressions handled by short-circuit evaluation.

Mapping Control Structures

Use templates, e.g.

```
while (Cond) { Stat1; Stat2; ... }
```

```
loop:
   MIPS code to check Cond; result in $t0
   beqz $t0, end_loop
   MIPS code for Stat1
   MIPS code for Stat2
   MIPS code for ...
   j    loop
end_loop:
```

... Mapping Control Structures

Template for `if...else if... else`

```
if (Cond1) Stat1 else if (Cond2) Stat2 else Stat3
```

```
if:
   MIPS code to check Cond1; result in $t0
   beqz $t0, else1
   MIPS code for Stat1
   j    end_if
else1:
   MIPS code to check Cond2; result in $t0
   beqz $t0, else2
   MIPS code for Stat2
   j    end_if
else2:
   MIPS code for Stat3
end_if:
```

Argc and Argv

The real MIPS machine has no idea about `argc` and `argv`

SPIM runs under Linux, and needs to interact with environment

So, the initialisation code (that invokes `main`) sets them up:

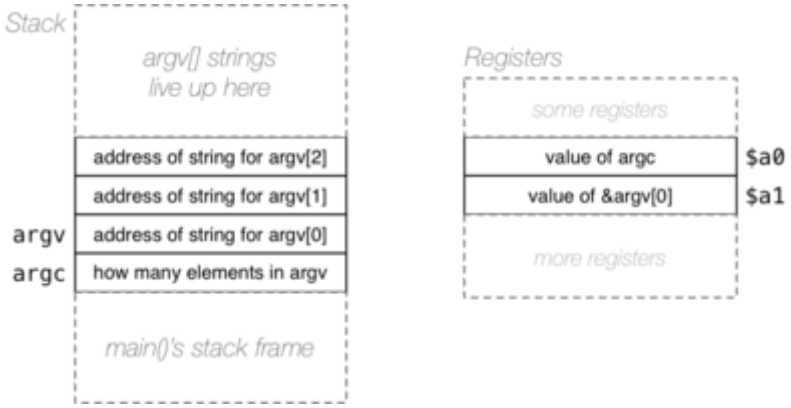
```
    lw    $a0 0($sp)   # argc
    addiu $a1 $sp 4    # argv
    ...
    jal   main
    nop
```

```
    li    $v0 10
    syscall           # syscall 10 (exit)
```

Note: we are ignoring `envp` (environment pointer)

... Argc and Argv

What the main program receives:



... Argc and Argv

Code to print the program's name (`argv[0]`):

```
# assume argc is $s0, argv is $s1
addi   $t0, $s1, 0    # &argv[0]
lw     $a0, ($t0)     # argv[]
li     $v0, 4
syscall
```

Code to print the first cmd-line arg (`argv[1]`)

```
# assume argc is $s0, argv is $s1
addi   $t0, $s1, 4    # &argv[4]
lw     $a0, ($t0)     # argv[]
li     $v0, 4
syscall
```

Produced: 2 Apr 2018