Computer Organization and Design

Kartik Mohanram

Department of Electrical and Computer Engineering
University of Pittsburgh
Pittsburgh, PA

kmram@pitt.edu

Spring 2018

Control, Loops, and Procedures

- 4 lectures span
  - Control (branching and jumps)
  - Loops (while and for)
  - Procedures
- Control
  - Instruction that potentially changes the flow of program execution
  - Conditional: bne, beq, and slt/blt instructions
  - Unconditional: j (jump) instruction
- Loops
- Procedures

Control

bne and beq

- bne (branch if not equal) and beq (branch if equal) instructions
- Both are I-format (more on this after a couple of examples)

Example: bne with $s0 ← i, $s1 ← j, $s2 ← k

```
if(i == j) k = i + j;
```

```
    bne $s0, $s1, LABEL
    add $s2, $s0, $s1
LABEL:
```

Example: bne and j with $s0 ← i, $s1 ← j, $s2 ← k

```
if(i == j) k = i + j;
    else k = i - j;
```

```
    bne $s0, $s1, ELSE
    add $s2, $s0, $s1
    j EXIT
ELSE: sub $s2, $s0, $s1
EXIT:
```

Control

slt and blt/bgt/ble/bge

- MIPS provides slt (set if less than)
  - Used to derive pseudo instruction blt (branch if less than)
  - Note that slt is an R-format instruction

Example: slt with $s0 ← i, $s1 ← j, $s2 ← k

if(i < j) k = 1
else k = 0;

```
slt $s2, $s0, $s1
```

- blt (branch if less than) is an I-format pseudo-instruction realized as a sequence of slt and bne instructions
- blt $s0, $s1, LABEL is equivalent to
  ```
  slt $at, $s0, $s1
  bne $at, $zero, LABEL
  ```
- $at is used by default to store the result of comparison
- Construct the bgt pseudo-instruction ...

Control

Addressing in I-format control instructions

- I-format: op $rt, $rs, IMM
31:26	25:21	20:16	15:0
op-code	rs	rt	IMM
- IMM is 16-bit signed (two's complement) off-set
  - Specifies number of instructions to be skipped (forward/backward)
- PC-relative addressing used; effective address computation:
- $PC_{new} \leftarrow PC_{old} + 4 + \text{sign-extend}(IMM \ll 2)$
  - Also denoted $PC_{new} \leftarrow PC_{old} + 4 + ((IMM_{15})^{14} \parallel (IMM \ll 2))$

Control

Addressing in I-format control instructions: Examples

Example: Offset calculation

```
0x000          beq $s0, $s1, LABEL
0x004          add $s2, $t0, $t1
0x008          ...
0x00C          ...
0x010          ...
0x014  LABEL:
```

$$PC_{new} \leftarrow PC_{old} + 4 + \text{sign-extend}(IMM \ll 2)$$

i.e., $0x014 = 0x000 + 4 + 4(IMM)$

i.e., $IMM = 4$

Example: Offset calculation

```
0x000  LABEL:  add $s0, $t0, $t1
0x004          add $s2, $t0, $t1
0x008          ...
0x00C          ...
0x010          ...
0x014          ...
0x018          beq $s0, $s1, LABEL
```

$$PC_{new} \leftarrow PC_{old} + 4 + \text{sign-extend}(IMM \ll 2)$$

i.e., $0x000 = 0x018 + 4 + 4(IMM)$

i.e., $IMM = -7$

Control

Immediate variants: slti

- MIPS provides slti (set if less than immediate)

Example: slti with $s0 ← i, $s1 ← j

```
if(i < 20) j = 1;          slti $s1, $s0, 20
   else j = 0;
```

- Pseudo-instruction bgti

Example: Realizing bgti with $s0 ← a

```
                           slti $at, $s0, 21
if(a > 20) a = a + 2;      bne $at, $zero, LABEL
                           addi $s0, $s0, 2
                  LABEL:   ...
```

- Construct pseudo-instruction blti
- Are immediate variants for beq and bne, i.e., beqi and bnei realizable?!

Control

jump instruction

- J-format: jump LABEL

31:26	25:0
op-code	IMM

- Construct 32-bit address from 26-bit IMM and PC

$$PC_{new} \leftarrow \{PC_{old}[31:28],IMM[25:0],00\}$$

- 256MB of addressable space for jumps ... how?

Control

Examples of control flow

- Annotated examples: please read the comments/annotations in the asm files to appreciate use of MIPS ISA
- weather.c and four asm variants
  - weather.asm — illustrates if-then-else and while loop
  - weather2.asm — a slightly improved version (still if-then-else)
  - weather3.asm — illustrates a "computed goto" (switch)
  - weather4.asm — illustrates an algorithm change, using a table
- sam12.asm — convert a 32-bit number into hex characters, which are displayed with the OS print string service

Control in action

while loops

- Comparison, branch to exit else work, and jump to continue

pseudo-code/template for while block

```
while (condition == true) {
    some work;
}
```

```
loop:   check condition
        branch if condition is false to exit
        some work
        j loop
exit:   ...
```

Example: Printing a string

```
char *str = "Hello World!";
char *s = str;
while (*s != '\0') {
    printf("%c", *s);
    s = s + 1;
}
```

```
        la $t0, str
loop:   lb $a0, 0($t0)
        beq $a0, $0, exit
        li $v0, 11 # print character
        syscall
        addi $t0, $t0, 1
        j loop
exit:   ...
```

Control in action

do while loops

- while loop uses one branch and one jump on each pass
  - Branches and jumps are expensive, especially with pipelining
  - Compilers can optimize this to at most one branch or one jump
  - do while loop illustration for print string routine
- Work, comparison, branch to continue else exit

pseudo-code/template for do while block

```
do { some work; }
   while (condition == true);
```

```
loop:   some work
        check condition
        branch to loop if condition is true
exit:   ...
```

Control in action

do while illustration

- Cleaner code, but corner case (null string) exists

Example: Printing a string

```
char *str = "Hello World!";
char *s = str;
/* breaks for null string */
do {
   printf("%c", *s);
   s = s + 1;
} while (*s != '\0')
```

```
        la $t0, str
        lb $a0, 0($t0) # start the loop
loop:   li $v0, 11 # print character
        syscall
        addi $t0, $t0, 1
        lb $a0, 0($t0)
        bne $a0, $0, loop
exit:   ...
```

- Handle corner case outside loop
  - Work through example printstring.c
  - Code looks hefty in C, but assembly trivial ... one conditional test outside loop and label to exit to

Procedures

Motivation

```
int len(char *s) {
  int l = 0;
  for(; *s != '\0'; s++) l++;
  return l;
}
void reverse(char *s, char *r) {
  char *p, *t;
  int l = len(s);
  *(r+l) = '\0';
  l--;
  for(p=s+l, t=r; l >= 0; l--) {
    *t++ = *p--;
  }
}
int main(void) {
  char *s = "Hello World!";
  char r[100];
  reverse(s, r);
  printf("%s\n", r);
}
```

- Aside: Pre-increment (++x) versus post-increment (x++)
- What is the output?!

```
int main(int argc, char **argp) {
  int i = 0, j = 0;
  for(i = 0; i < 5; i = j++) {
    printf("%d\n", i);
  }
  j = 0;
  for(i = 0; i < 5; i = ++j) {
    printf("%d\n", i);
  }
}
```

- Google C++ Style guide: Use prefix form (++i) of the increment and decrement operators with iterators and other template objects (more to it than meets the eye)

Procedures

Call and return

- Caller: A procedure that calls another procedure
- Callee: The procedure that is called by the caller
- Procedure call
  - Jump to the procedure
  - Ensure return to the point immediately after the call
  - Need to pass "return address", i.e., the address of the instruction after the call to callee
  - jal LABEL has the following effect
    - \$ra $\leftarrow$ PC + 4, i.e., \$ra holds the return address
    - $PC_{new} \leftarrow PC_{old}[31:28] \mid (LABEL \ll 2)$, i.e., a jump to the procedure is initiated
- Procedure return
  - Need to restore return address to PC to continue execution in caller
  - jr \$ra has the following effect
    - $PC_{new} \leftarrow$ \$ra, i.e., a jump back to the return address is initiated

Procedures

Arguments, return values, call chains

- Conventions specified in PRM
  - $a0-$a3: four arguments for passing values to callee
  - $v0-$v1: two values returned by callee to caller
  - $ra: return address register (set by caller, used on return from callee)
- Call chains
  - One procedure calls another, which calls another
  - E.g., main $\rightarrow$ reverse $\rightarrow$ len
  - What happens to $ra??? (e.g., when reverse calls len)
- You must save $ra someplace!
  - Simple approach: A "free" register (cannot be used by caller)
  - Leaf procedure: Does not make any calls; does not need to save $ra

Procedures

Registers

- What if callee wants to use registers?
  - Caller is also using registers!!!
  - If callee wants to use same registers, it must save them
  - Consider what happened with $ra in a call chain
- Register usage conventions specified in PRM
  - $t0-$t9: Temporary/scratch registers; if caller wants to use them after the call, they must be saved before the call
  - $s0-$s7: Saved registers; must be saved by callee prior to using them
- So far so good, but caller/callee need memory space to hold saved ("spilled") registers
  - Caller spills $t0-$t9 that be must saved to memory
  - Callee spills $s0-$s7 to memory, when these registers are used
  - Other registers (e.g., $v0, $v1 may also need to be saved)
  - Non-leaf caller saves $ra before making another call
- Each procedure needs memory space to save registers
- Call-chain depth (number of called procedures) unknown, need to support undetermined length
- Solution: Use a stack located in memory.

Procedures

Program stack

- Program stack: Memory locations used by running program
  - Has space for saved registers
  - Has space for local variables, when they cannot all fit in registers
    - E.g., local arrays are allocated on the stack
  - Has space for return address
  - Stack big-picture concepts from the textbook:

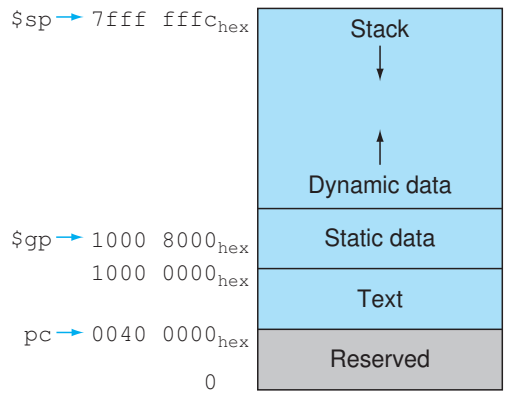

FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to $7fff\ fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at $0040\ 0000_{hex}$. The static data starts at $1000\ 0000_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, `$gp`, is set to an address to make it easy to access data. It is initialized to $1000\ 8000_{hex}$ so that it can access from $1000\ 0000_{hex}$ to $1000\ ffff_{hex}$ using the positive and negative 16-bit offsets from `$gp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Procedures

Program stack

- Each procedure explicitly allocates space for these items
  - So-called activation frame (a.k.a. activation record)
  - Purpose of locations in activation frame are known
  - Location of activation frame is not known until procedure call
- Prologue (entry point into the procedure): Allocates an activation frame on the stack
- Epilogue (exit point from procedure): De-allocates the activation frame, does actual return
- PRM specifies use of two registers:
  - $sp: pointer to top of stack
  - $fp: pointer to top of current activation frame

Procedures

Program stack: Conventions for use

- Caller: save needed registers on stack, set up arguments, make call
  - If arguments exceed 4, place arguments on stack
- Callee prologue
  - Allocate space on stack for saved registers, locals, return address (if callee is a non-leaf procedure)
  - Save registers and return address
- Callee procedure body
  - Access stack as necessary, including loading arguments
- Callee epilogue
  - Restore return address from stack (if callee is a non-leaf procedure)
  - Restore saved registers
  - Ensure return values are in $v0 and $v1
  - Return to caller
- Comprehensive example: factorial

Procedures

Program stack: Conventions for use

- Prologue-epilogue illustration from the textbook:

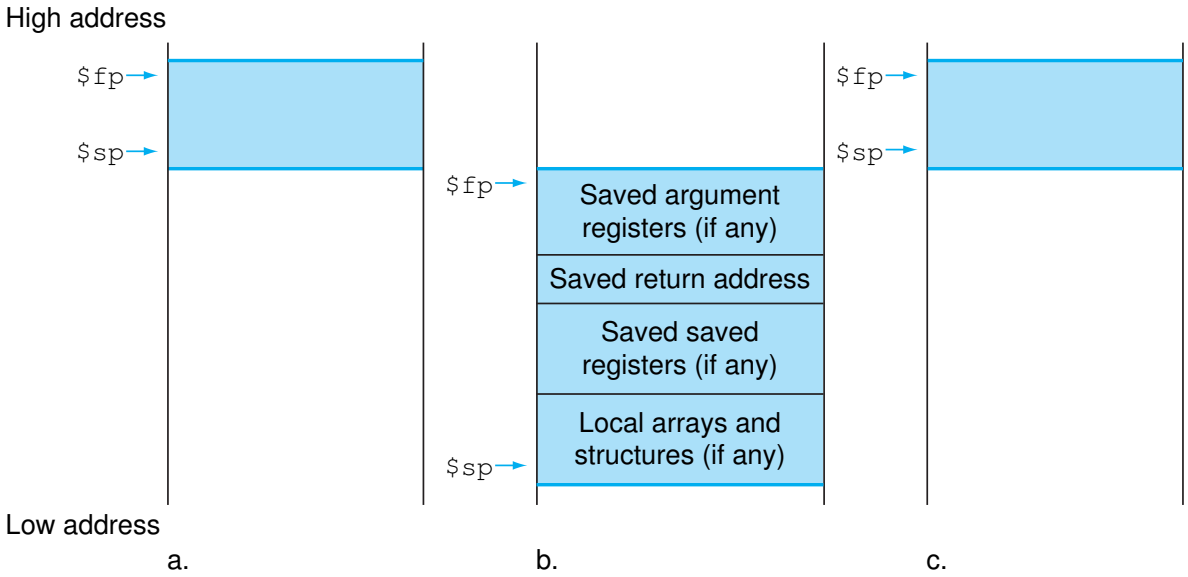

FIGURE2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`$fp`) points to the first word of the frame, often a saved argument register, and the stack pointer (`$sp`) points to the top of the stack. The stack is adjusted to make room for all the saved registers and any memory-resident local variables. Since the stack pointer may change during program execution, it's easier for programmers to reference variables via the stable frame pointer, although it could be done just with the stack pointer and a little address arithmetic. If there are no local variables on the stack within a procedure, the compiler will save time by *not* setting and restoring the frame pointer. When a frame pointer is used, it is initialized using the address in `$sp` on a call, and `$sp` is restored using `$fp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Miscellanea

MIPS addressing modes

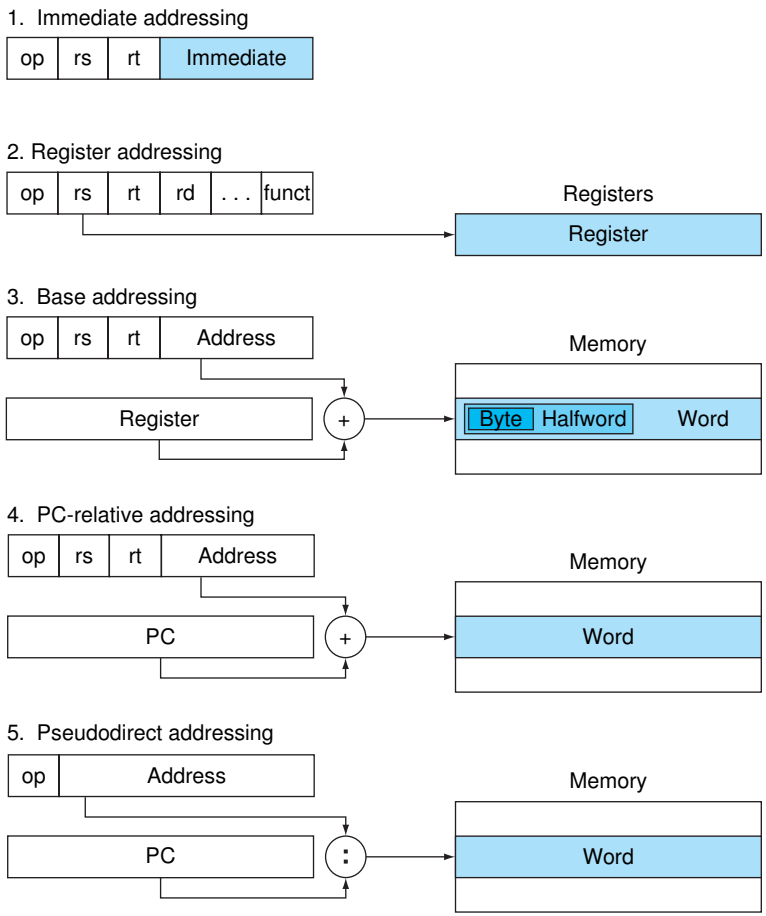

FIGURE2.18 Illustration of the five MIPS addressing modes. The operands are shaded in color. The operand of mode 3 is in memory, whereas the operand for mode 2 is a register. Note that versions of load and store access bytes, halfwords, or words. For mode 1, the operand is 16 bits of the instruction itself. Modes 4 and 5 address instructions in memory, with mode 4 adding a 16-bit address shifted left 2 bits to the PC and mode 5 concatenating a 26-bit address shifted left 2 bits with the 4 upper bits of the PC.

- Immediate addressing, where the operand is a constant within the instruction
- Register addressing, where the operand is a register
- Base or displacement addressing, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction
- PC-relative addressing, where the address is the sum of the PC and a constant in the instruction
- Pseudodirect addressing, where the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Miscellanea

ASCII representation

ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character	ASCII value	Character
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

FIGURE 2.15 ASCII representation of characters. Note that upper- and lowercase letters differ by exactly 32; this observation can lead to shortcuts in checking or changing upper- and lowercase. Values not shown include formatting characters. For example, 8 represents a backspace, 9 represents a tab character, and 13 a carriage return. Another useful value is 0 for null, the value the programming language C uses to mark the end of a string. This information is also found in Column 3 of the MIPS Reference Data Card at the front of this book.

```
   .text
      li    $a0, 30     # compute 5!
      jal _fact
      move  $a0, $v0     # get result
      li    $v0, 1      # print integer
      syscall
      move  $a0, $v1    # get result
      li    $v0, 1      # print integer
      syscall
      li    $v0, 10
      syscall
      #
      # fact(arg) - computes factorial of arg (arg!)
      # argument is passed in $a0
      # stack frame:
      #
      #       | ...high address... |
      #       |--------------------|
      #       |                    |
      #       |--------------------|
      #       |  return address    |  +4
      #       |--------------------|
      #  $sp->|       saved $s0    |  +0
      #       |--------------------|
      #       |  ...low address... |
      #
_fact:
    # prologue to procedure
    addi    $sp,$sp,-8  # push space for activation frame
    sw  $s0,0($sp)     # save $s0, which we use
    sw  $ra,4($sp)     # save return address
    # start of actual procedure work
    move    $s0,$a0     # get argument ($a0)
    li  $v0,0x1         # 1
    beq $s0,$v0,_fact_exit  # end of recursion (f==1?)
    addi    $a0,$s0,-1  # f /= 1, so continue. set up arg(f-1)
    jal _fact           # recursive call
    mult    $v0,$s0     # multiply
    mflo    $v0         # return mul result
    mfhi    $v1
_fact_exit:
    # epilogue to exit procedure
    lw  $ra,4($sp)      # restore $ra
    lw  $s0,0($sp)      # restore $s0
    addi    $sp,$sp,8   # pop activation frame
    jr  $ra             # return
```