Computer Architecture

The Language of the Machine

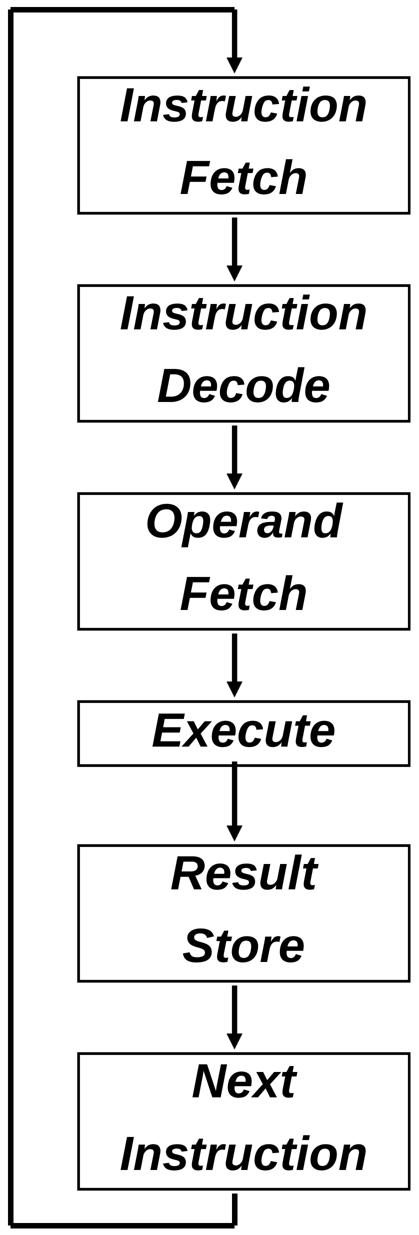
Instruction Sets

- Basic ISA
- Classes, Addressing, Format
- Administrative Matters
- Operations, Branching, Calling conventions
- Break

Organization

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all “memory” are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory--we can have more
- Input and output (I/O) devices have the messiest organization
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ...
  - Least amount of research (so far)

Instruction Set Architecture: What Must be Specified?



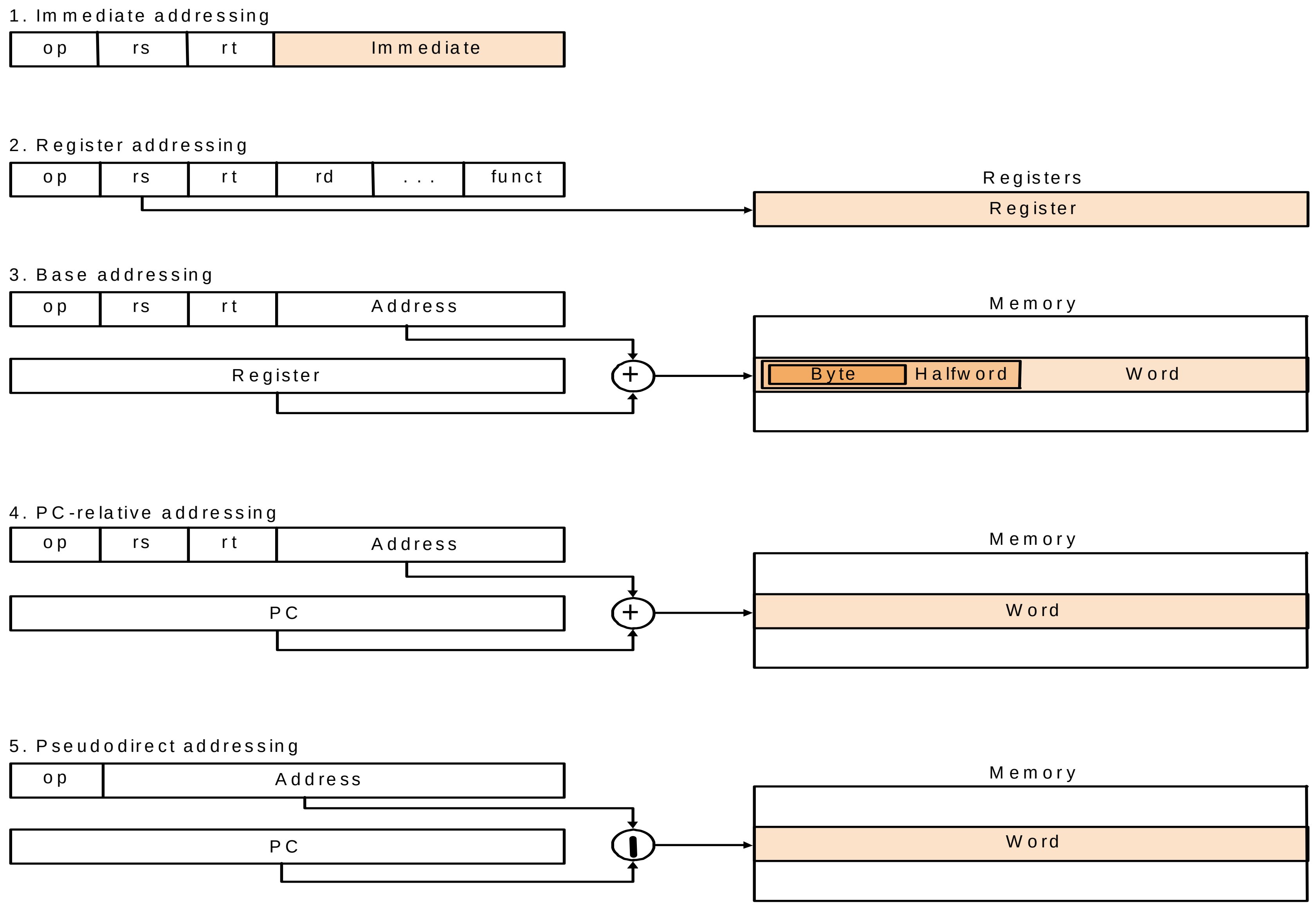
° Instruction Format or Encoding
  – how is it decoded?
° Location of operands and result
  – where other than memory?
  – how many explicit operands?
  – how are memory operands located?
  – which can or cannot be in memory?
° Data type and Size
° Operations
  – what are supported
° Successor instruction
  – jumps, conditions, branches
  - fetch-decode-execute is implicit!

RISC features

- Reduced Instruction Set
- General Purpose Register File (large number: 32 or more)
- Load/Store Architecture
- Few Addressing modes
- Fixed Instruction Format

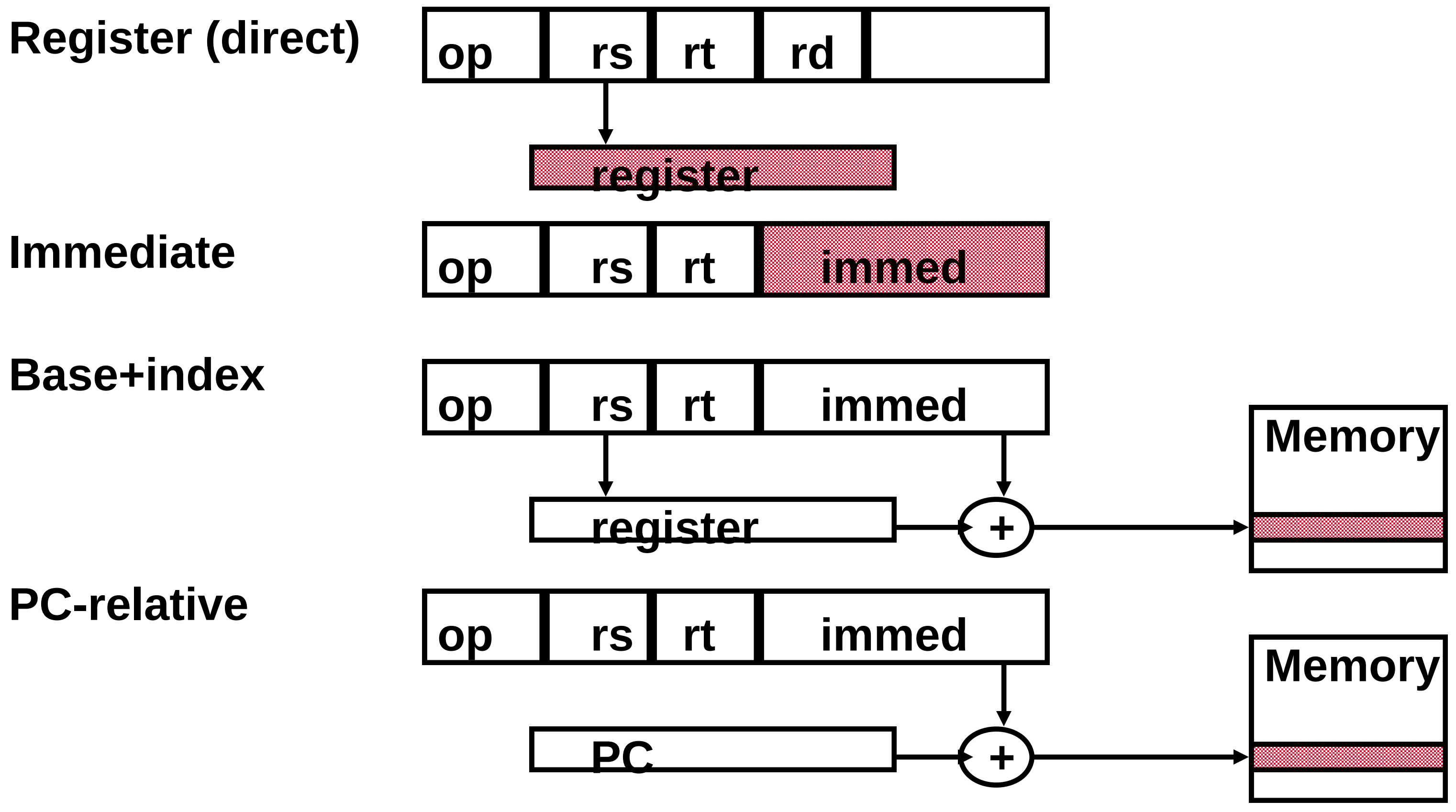
MIPS Addressing Formats (*Summary*)

- **How memory can be addressed in MIPS**



MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



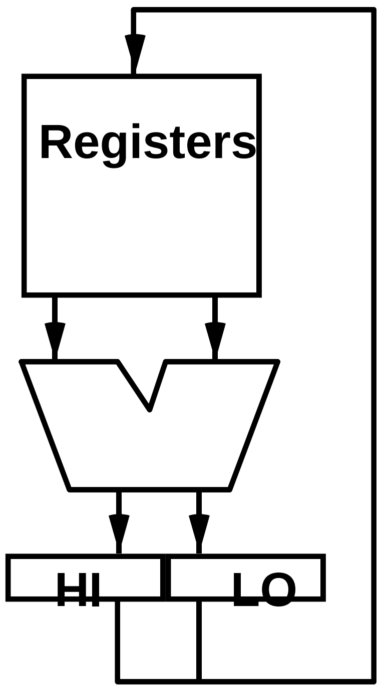
- Register Indirect?

MIPS I Operation Overview

- Arithmetic logical
    - Add, AddU, Sub,   SubU, And, Or, Xor, Nor, SLT, SLTU
    - AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
    - SLL, SRL, SRA, SLLV, SRLV, SRAV
- Memory Access
    - LB, LBU, LH, LHU, LW, LWL,LWR
    - SB, SH, SW, SWL, SWR

Multiply / Divide

- Start multiply, divide
  - MULT rs, rt
  - MULTU rs, rt
  - DIV rs, rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd
  - MFLO rd
- Move to HI or LO
  - MTHI rd
  - MTLO rd
- Why not Third field for destination?
  (Hint: how many clock cycles for multiply or divide vs. add?)



Data Types

<u>Bit</u>: 0, 1

<u>Bit String</u>: sequence of bits of a particular length

4 bits is a nibble
8 bits is a byte
16 bits is a half-word
32 bits is a word
64 bits is a double-word

<u>Character</u>:

ASCII 7 bit code

<u>Decimal</u>:

digits 0-9 encoded as 0000b thru 1001b
two decimal digits packed per 8 bit byte

<u>Integers</u>:

2's Complement

<u>Floating Point</u>:

Single Precision
Double Precision
Extended Precision

$$\underset{\text{mantissa}}{M} \times \underset{\text{base}}{R}^{\overset{\text{exponent}}{E}}$$

How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; exception possible
subtract	sub \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; exception possible
add immediate	addi \$1,\$2,100	\$1 = \$2 + 100	+ constant; exception possible
add unsigned	addu \$1,\$2,\$3	\$1 = \$2 + \$3	3 operands; no exceptions
subtract unsigned	subu \$1,\$2,\$3	\$1 = \$2 – \$3	3 operands; no exceptions
add imm. unsign.	addiu \$1,\$2,100	\$1 = \$2 + 100	+ constant; no exceptions
multiply	mult \$2,\$3	Hi, Lo = \$2 x \$3	64-bit signed product
multiply unsigned	multu\$2,\$3	Hi, Lo = \$2 x \$3	64-bit unsigned product
divide	div \$2,\$3	Lo = \$2 ÷ \$3, Hi = \$2 mod \$3	Lo = quotient, Hi = remainder
divide unsigned	divu \$2,\$3	Lo = \$2 ÷ \$3, Hi = \$2 mod \$3	Unsigned quotient & remainder
Move from Hi	mfhi \$1	\$1 = Hi	Used to get copy of Hi
Move from Lo	mflo \$1	\$1 = Lo	Used to get copy of Lo

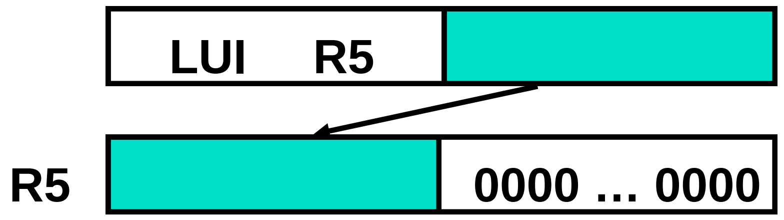
Which add for address arithmetic? Which add for integers?

MIPS logical instructions

Instruction	***Example***	***Meaning***	***Comment***
and	and \$1,\$2,\$3	\$1 = \$2 & \$3	3 reg. operands; Logical AND
or	or \$1,\$2,\$3	\$1 = \$2 | \$3	3 reg. operands; Logical OR
xor	xor \$1,\$2,\$3	\$1 = \$2 ⊕ \$3	3 reg. operands; Logical XOR
nor	nor \$1,\$2,\$3	\$1 = ~(\$2 |\$3)	3 reg. operands; Logical NOR
and immediate	andi \$1,\$2,10	\$1 = \$2 & 10	Logical AND reg, constant
or immediate	ori \$1,\$2,10	\$1 = \$2 | 10	Logical OR reg, constant
xor immediate	xori \$1, \$2,10	\$1 = ~\$2 &~10	Logical XOR reg, constant
shift left logical	sll \$1,\$2,10	\$1 = \$2 << 10	Shift left by constant
shift right logical	srl \$1,\$2,10	\$1 = \$2 >> 10	Shift right by constant
shift right arithm.	sra \$1,\$2,10	\$1 = \$2 >> 10	Shift right (sign extend)
shift left logical	sllv \$1,\$2,\$3	\$1 = \$2 << \$3	Shift left by variable
shift right logical	srlv \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right by variable
shift right arithm.	srav \$1,\$2, \$3	\$1 = \$2 >> \$3	Shift right arith. by variable

MIPS data transfer instructions

Instruction	*Comment*
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)



Why need LUI?

Methods of Testing Condition

° Condition Codes

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

```
ex:     add r1, r2, r3
        bz label
```

° Condition Register

```
Ex:     cmp r1, r2, r3
        bgt r1, label
```

° Compare and Branch

```
Ex:     bgt r1, r2, label
```

MIPS Compare and Branch

- Compare and Branch
  - BEQ rs, rt, offset   if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset   <>
- Compare to zero and Branch
  - BLEZ rs, offset   if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset   >
  - BLT   <
  - BGEZ   >=
  - BLTZAL rs, offset   if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL   >=
- Remaining set of compare and branch take two instructions
- Almost all comparisons are against zero!

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq \$1,\$2,100	if (\$1 == \$2) go to PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne \$1,\$2,100	if (\$1!= \$2) go to PC+4+100
	Not equal test; PC relative	
set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0
	Compare less than; 2's comp.	
set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0
	Compare less than; natural numbers	
set l. t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0
	Compare < constant; natural numbers	
jump	j 10000	go to 10000
	Jump to target address	
jump register	jr \$31	go to \$31
	For switch, procedure return	
jump and link	jal 10000	\$31 = PC + 4; go to 10000
	For procedure call	

Signed vs. Unsigned Comparison

Value?

2's comp   Unsigned?

R1= `0...00 0000 0000 0000 0001` two

R2= `0...00 0000 0000 0000 0010` two

R3= `1...11 1111 1111 1111 1111` two

- After executing these instructions:

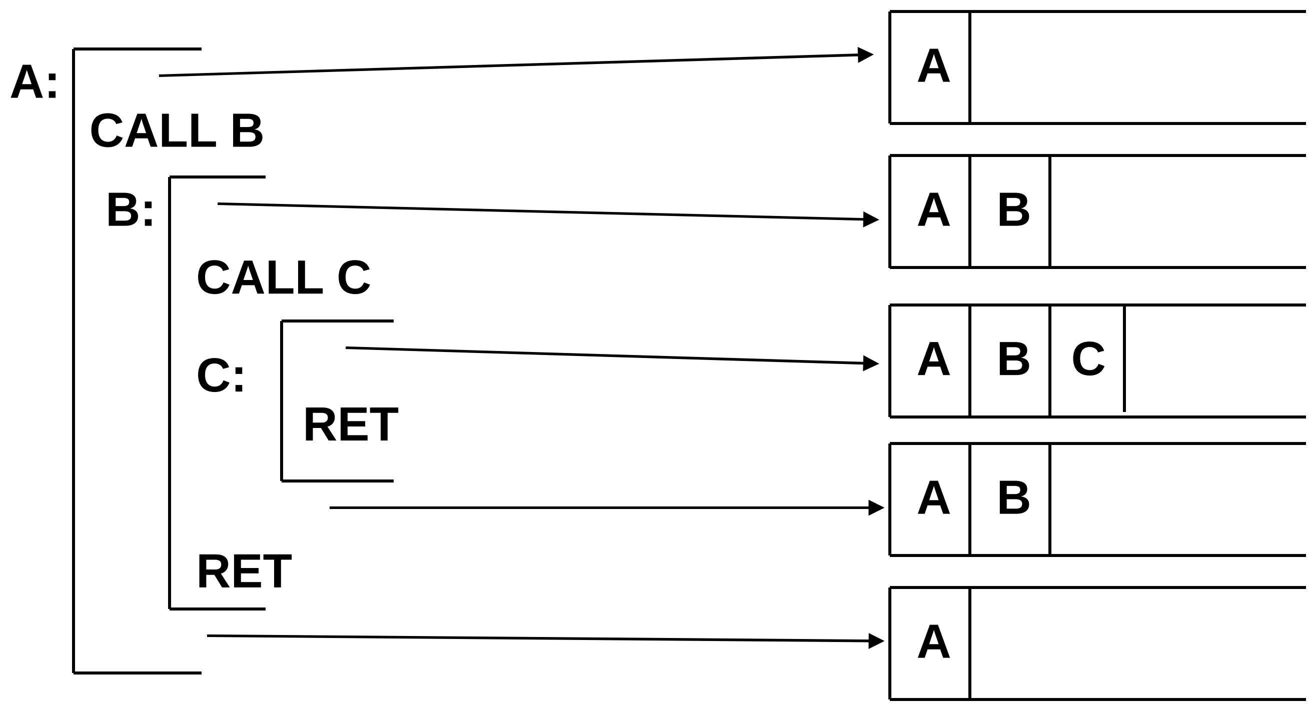
```
slt  r4,r2,r1 ; if (r2 < r1) r4=1; else r4=0
slt  r5,r3,r1 ; if (r3 < r1) r5=1; else r5=0
sltu r6,r2,r1 ; if (r2 < r1) r6=1; else r6=0
sltu r7,r3,r1 ; if (r3 < r1) r7=1; else r7=0
```

- What are values of registers r4 - r7? Why?

r4 =   ; r5 =   ; r6 =   ; r7 =   ;

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



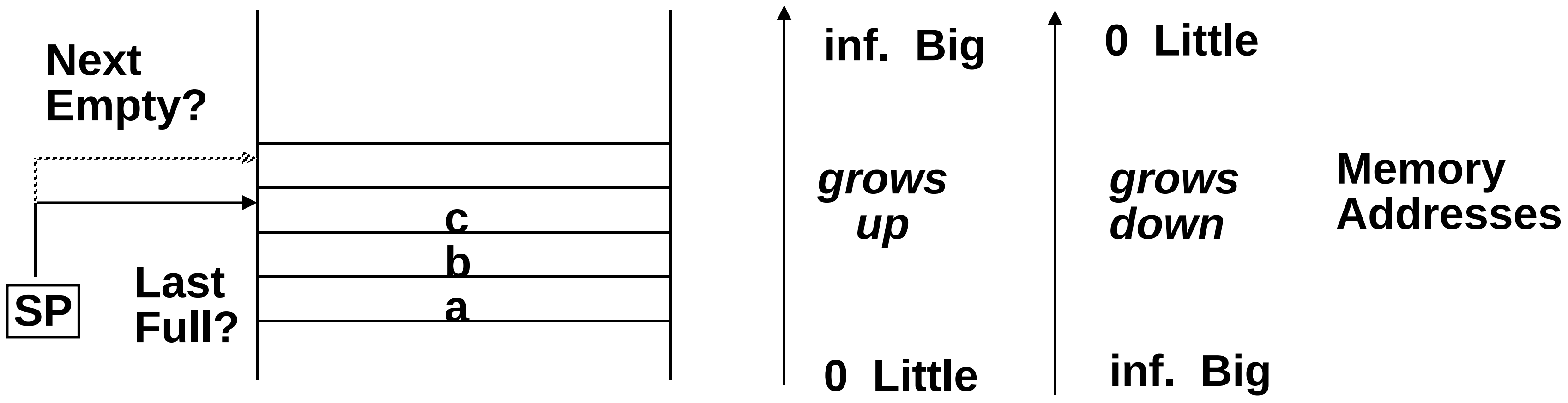
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
Decrement SP

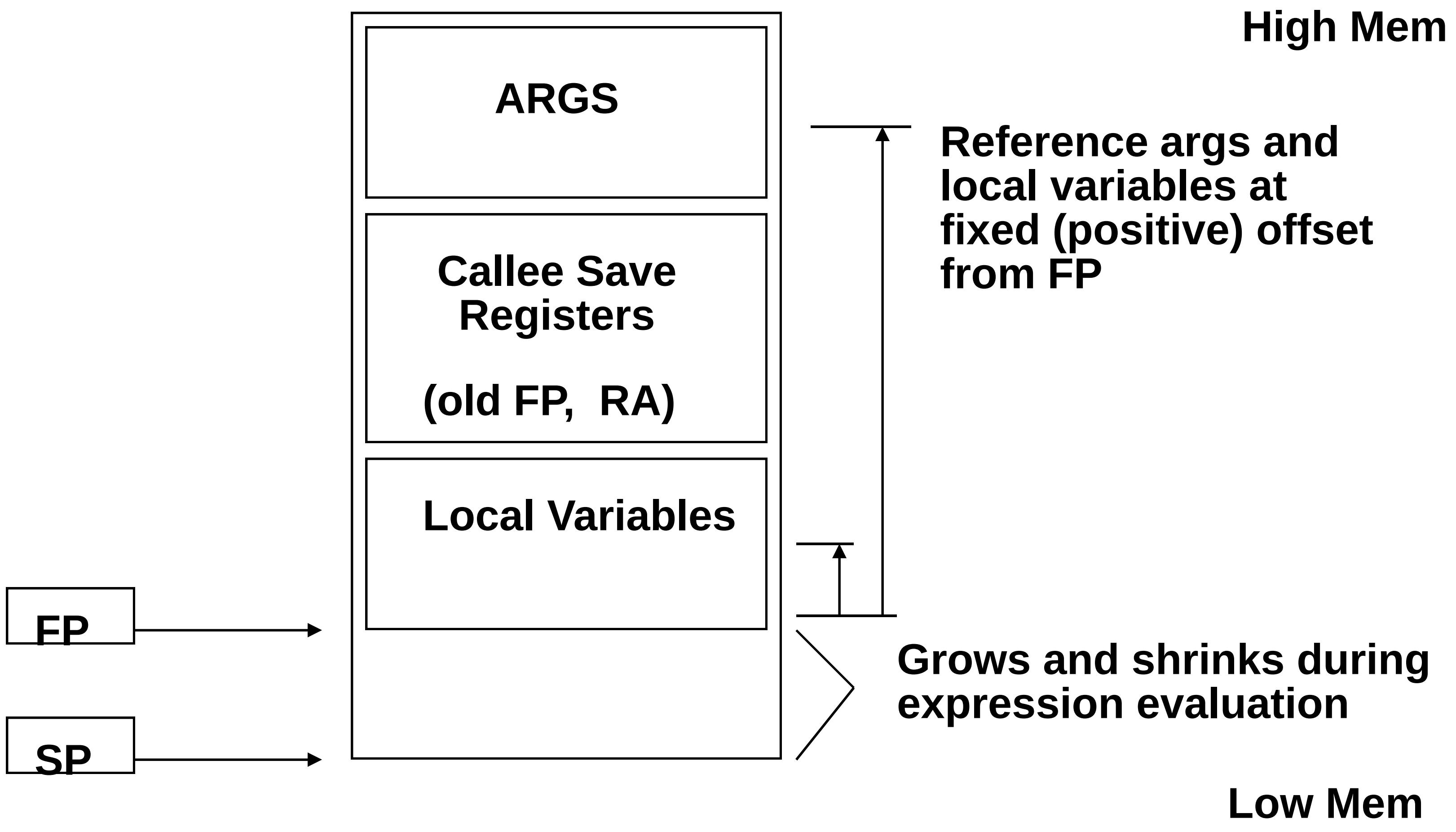
PUSH: Increment SP
Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
Read from Mem(SP)

PUSH: Write to Mem(SP)
Increment SP

Call-Return Linkage: Stack Frames


High Mem
ARGS
Callee Save Registers
(old FP, RA)
Local Variables
Reference args and local variables at fixed (positive) offset from FP
FP
SP
Grows and shrinks during expression evaluation
Low Mem

- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame
- Compilers normally keep scalar variables in registers, not memory!

MIPS: Software conventions for Registers

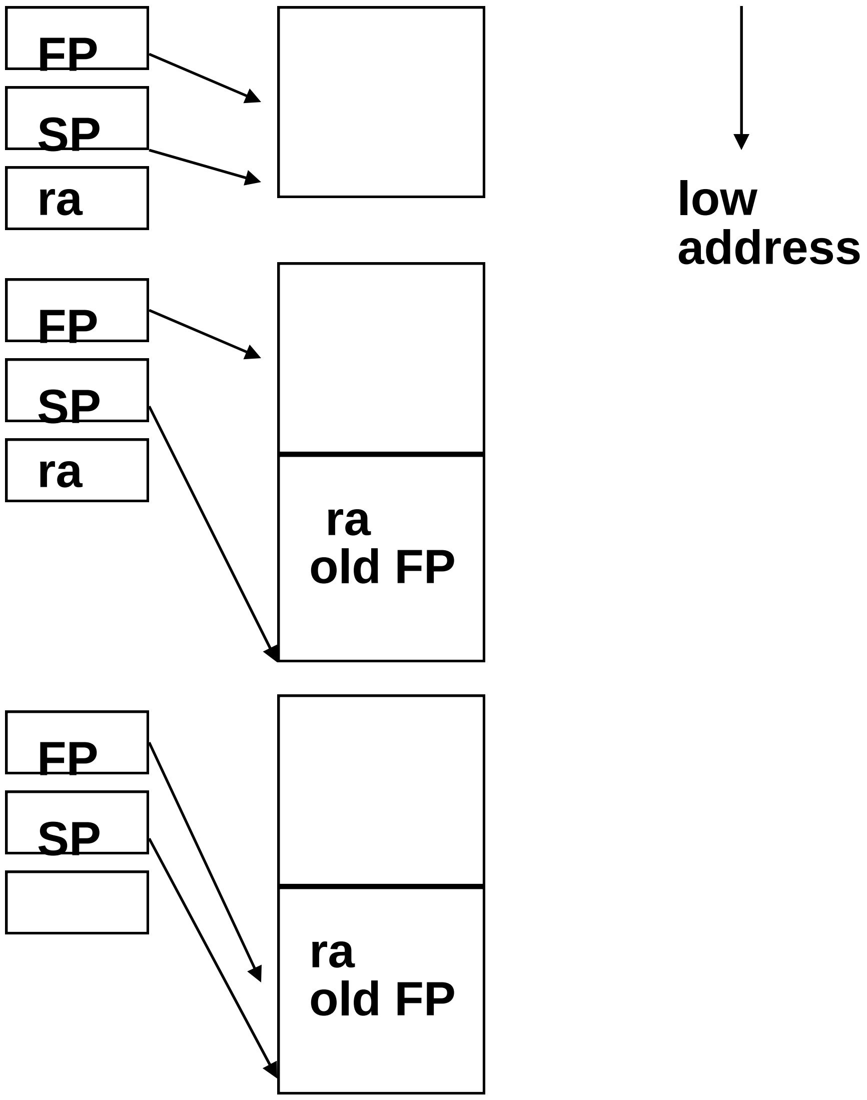
0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	

16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	Return Address (HW)

Plus a 3-deep stack of mode bits.

MIPS / GCC Calling Conventions

```
fact:
   addiu        $sp, $sp, -32
   sw     $ra, 20($sp)
   sw     $fp, 16($sp)
   addiu  $fp, $sp, 32
. . .
   sw     $a0, 0($fp)
...
   lw     $31, 20($sp)
   lw     $fp, 16($sp)
   addiu  $sp, $sp, 32
   jr     $31
```



First four arguments passed in registers.

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Branch/jump and link put the return addr. PC+4 into the link register (R31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it cannot occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

MIPS Instructions (Quick Summary)

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Summary of RISC

- Reduced Instruction Set
- General Purpose Register File (large number: 32 or more)
- Load/Store Architecture
- Few Addressing modes
- Fixed Instruction Format

MIPS Architecture

- 32 Registers
- Load/Store Architecture
- 5 Instruction Groups: Arithmetic, Logical, Data Transfer, Cond. Branch, Uncond. Jump
- Addressing modes: Register, Displacement, Immediate and PC-relative
- Fixed Instruction Format

Registers

- General Purpose Register Set
- Any register can be used with any instruction
- MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code.

Registers

Symbolic Name	Number	Usage
zero	0	Zero
at	1	Reserved for the Assembler
v0 – v1	2 - 3	Result Registers
a0 – a3	4 - 7	Argument Registers 1…4
t0 – t9	8 – 15, 24 - 25	Temporary Registers 0…9
s0 – s7	16 - 23	Saved Registers 0…7
k0 – k1	26 - 27	Kernel Registers 0…1
gp	28	Global Data Pointer
sp	29	Stack Pointer
fp	30	Frame Pointer
ra	31	Return Address

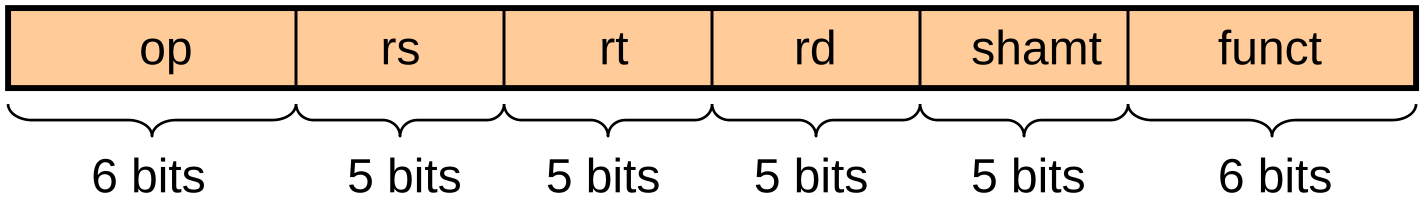
Instruction Format

- Fixed Format
- 3 Format Types
    - Register: R-type
    - Immediate: I-type
    - PC-relative: J-type

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

All MIPS Instructions Format

R-Type



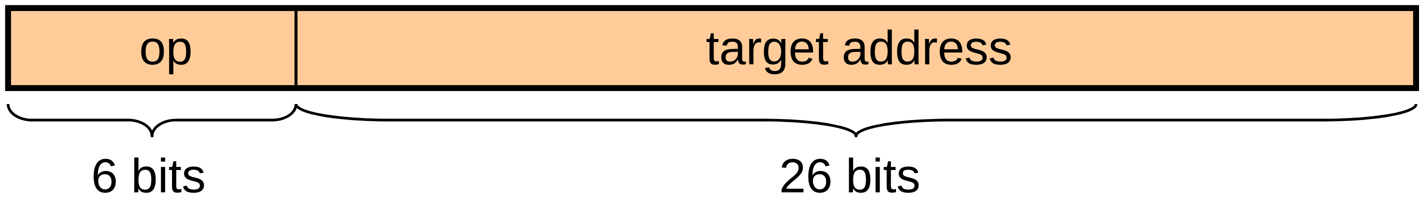
° **Used by**
- **Arithmetic Instructions**
- **Logic Instructions**
- **Except when Immediate Addressing mode used**

I-Type

op	rs	rt	address/immediate
6 bits	5 bits	5 bits	16 bits

- **Used by**
  - **Instructions using Immediate addressing mode**
  - **Instructions using Displacement addressing mode**
  - **Branch instructions**

J-Type



- Used by
  - Jump Instructions

Instructions

- 5 Groups
    - Arithmetic
    - Logic
    - Data Transfer
    - Conditional Branch
    - Unconditional Jump

Arithmetic

- add, addu: signed and unsigned addition on registers
- addi, addiu: signed and unsigned addition. One operand is immediate value
- sub, subu: signed and unsigned subtraction on registers
- subi, subiu: signed and unsigned subtraction. One operand is immediate value
- mult, multu: signed and unsigned multiplication on registers
- div, divu: signed and unsigned division on registers
- mfc0: move from coprocessor
- mfhi, mflo: move from Hi and Lo registers

Logical

- and, andi: logical ‘AND’ on registers and registers and an immediate value
- nor, nori: logical ‘NOT OR’ on registers and registers and an immediate value
- or, ori: logical ‘OR’ on registers and registers and an immediate value
- xor, xori: logical ‘Exclusive OR’ on registers and registers and an immediate value
- sll, sra, srl: shift left/right logical/arithmetic on registers. Size of shift can be immediate value.
- slt: comparison instruction: rd $\leftarrow$ 1/0 depending on comparison outcome

Data Transfer

- lw, sw: load/store word
- lb, sb: load/store byte
- lbu: load byte unsigned
- lh, sh: load/store halfword
- lui: load upper half word immediate

Branch

- b: branch unconditional
- beq: branch if src1 == src2
- bne: branch if src1 =/= src2
- bgez: branch is src1 >= 0
- bgtz: branch if src1 > 0
- blez: branch if src1 <= 0
- bltz: branch if src1 < 0

Jump

- j: jump
- jr: jump to src1 (address in reg src1)
- jal: jump and link; ra $\leftarrow$ PC+4; jump to label
- jalr: jump and link; ra $\leftarrow$ PC+4; jump to src1 (address in reg src1)

Addressing Modes

- Register: all operands are registers
- Immediate: one operand is an immediate value contained in the immediate field of I-type format
- Displacement: The address of the operand is src1 + displacement. Also contained in the immediate field of I-type format
- PC-relative: The +/- displacement is sign extended and added to the PC
- Direct Address: used by jump instructions. The full address is provided.