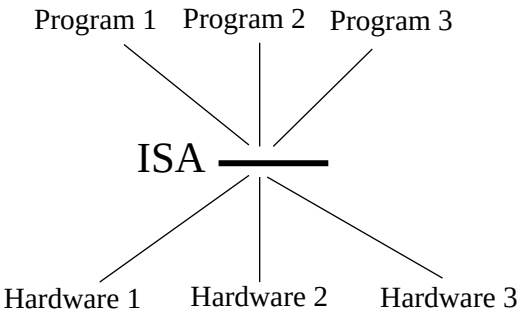
Lectures 5-6: Instruction Set Architectures - II

- Announcements
  - Undergrad career fair
    - Thursday 10am-3pm, http://careers.ns.utexas.edu
  - Homework #1 due now
  - HW #2 will be posted online ASAP
  - Pair programming for HW #2
- Last Time
  - Introduction to ISAs
- Today
  - Instruction types
  - Memory operations
  - Control flow operations
  - Instruction formats
  - Case study ISAs – MIPS, others

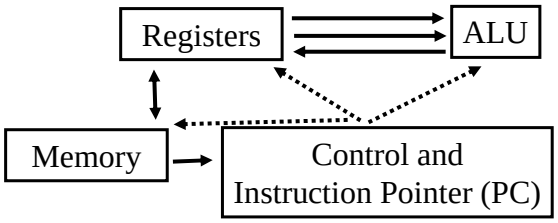
Pair Programming

- Two-person programming teams
- Work side-by-side
  - One person "drives" (types the code)
  - Other person watches, thinks, and makes suggestions
  - Two brains are better than one
- One grade per team
- Pick your own partner
  - Find someone with similar skill level as you
  - And a compatible schedule
  - OK to change after this assignment
- Issues to be aware of:
  - Both partners must learn; take turns driving
  - It takes time to get used to this programming method

ISA is an interface (abstraction layer)



A typical ISA machine model

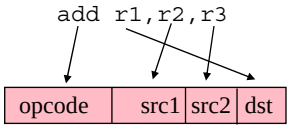


Architecture vs. Implementation

- Architecture: defines what a computer system does in response to a program and a set of data
  - Programmer visible elements of computer system
- Implementation: defines how a computer does it
  - Sequence of steps to complete operations
  - Time to execute each operation
  - Hidden "bookkeeping" functions

Components of Instructions

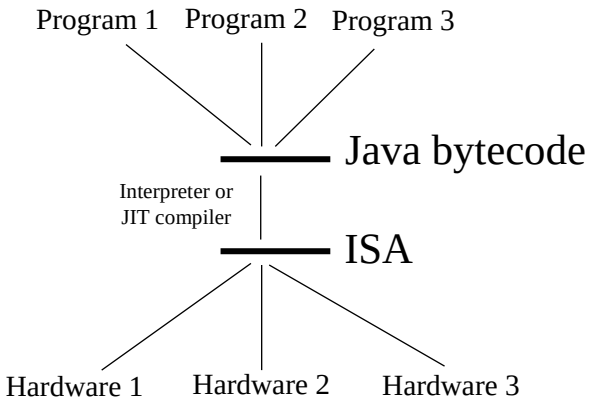
- Operations (opcodes)
- Number of operands
- Operand specifiers



- Instruction encodings
- Instruction classes
  - ALU ops (add, sub, shift)
  - Branch (beq, bne, etc.)
  - Memory (ld/st)

What about Java and JVM?

JVM is an additional interface layer between program and hardware ISA.
Microsoft's .NET common language runtime (CLR) has this same role.



ISA's in Detail

Instruction Types

- **ALU Operations**
  - arithmetic (`add, sub, mult, div`)
  - logical (`and, or, xor, srl, sra`)
  - data type conversions (`cvtf2d, cvtf2i`)
- **Data Movement**
  - memory reference (`lb, lw, sb, sw`)
  - register to register (`movi2fp, movf`)
- **Control** - what instruction to do next
  - tests/compare (`slt, seq`)
  - branches and jumps (`beq, bne, j, jr`)
  - support for procedure call (`jal, jalr`)
  - operating system entry (`trap`)
- **Hair** - string compare!

Addressing Modes
Driven by Program Usage

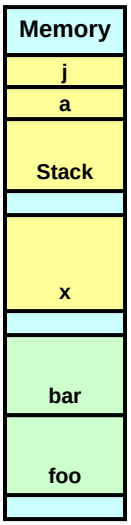
```
double x[100] ;          // global
void foo(int a) {        // argument
  int j ;                // local
  for(j=0;j<10;j++)
    x[j] = 3 + a*x[j-1] ;
  bar(a);
}
```

procedure

constant

argument

array reference

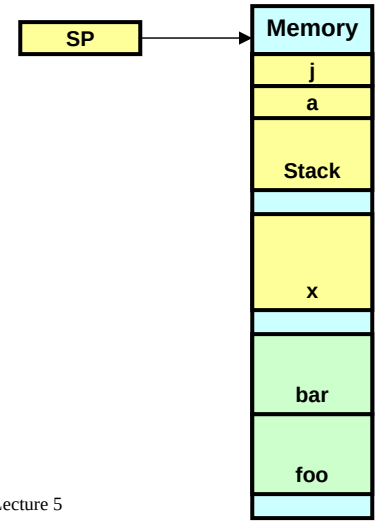


Assembly language example

```
double x[100] ;
void foo(int a) {
  int j;
  for(j=0;j<10;j++)
    x[j] = 3 + a*x[j-1];
  bar(a);
}
```


Addressing Modes

- Stack relative for locals and arguments
  a, j: *(R30+x)
- Short immediates (small constants)
  3
- Long immediates (global addressing)
  &x[0], &bar: 0x3ac1e400
- Indexed for array references
  *(R4+R3)



Addressing Mode Summary

`#n`	immediate
`(0x1000)`	absolute
`Rn`	Register
`(Rn)`	Register indirect
`-(Rn)`	predecrement
`(Rn)+`	postincrement
`*(Rn)`	Memory indirect
`*(Rn)+`	postincrement
`d(Rn)`	Displacement (b,w,l)
`d(Rn)[Rx]`	Scaled

VAX 11 had 27 addressing modes (why?)

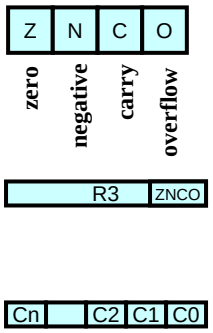
Control Instructions

- Implicit control on each instruction
  - PC ← PC + 4
- Unconditional jumps
  - PC ← X (direct)
  - PC ← PC + X (PC relative)
  - X can be constant or register
- Conditional jumps (branches)
  - PC ← PC + ((cond) ? X : 4)
- Predicated instructions
- Conditions
  - flags
  - in a register
  - fused compare and branch

```
LOOP:   LOAD    R1 <- (R5+R2)
        ADD     R3 <- R3 + R1
        ADD     R2 <- R2 + 4
        CMP     R4 <- R2 == 8
        JNE     R4, LOOP
```

Conditional Branching

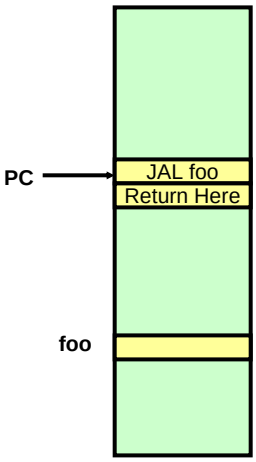
- Compute condition first
  - Condition codes
    ```
    CMP R1, R2
    BGE LOOP
    ```
    - Forces CMP and BR to be adjacent
  - Condition in GP register
    ```
    CMP R3, R1, R2
    BGE R3, LOOP
    ```
    - Enables parallelism of comparisons
  - Condition in "condition" register



- Fuse condition check and branch
  ```
  BGE R1, R2, LOOP
  ```
  - reduces instruction count, but complicates pipelining

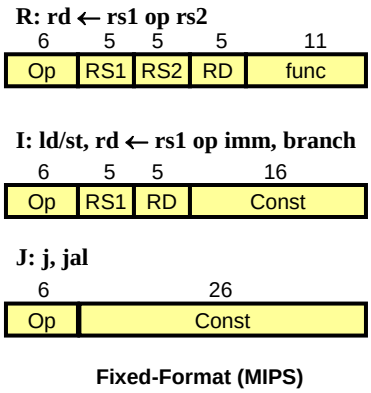
Support for Procedures

- Branch and Link
  - store return address in reg and jump
    JALR Rdest: Rx ← PC + 4, PC ← Dest
- Subroutine call
  - push return address on stack and jump
- CALLP (VAX)
  - push return address
  - set up stack frame
  - save registers
  - ...



Instruction Formats

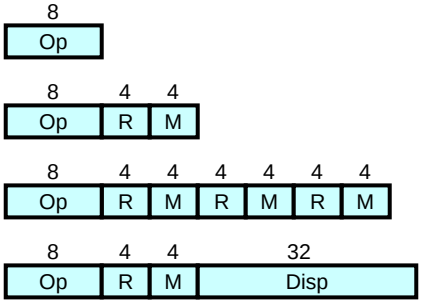
- Different instructions need to specify different information
  - return
  - increment R1
  - R3 ← R1 + R2
  - jump to 64-bit address
- Frequency varies
  - instructions
  - constants
  - registers
- Can encode
  - fixed format
  - small number of formats
  - byte/bit variable



Fixed-Format (MIPS)

Variable-Length Instructions

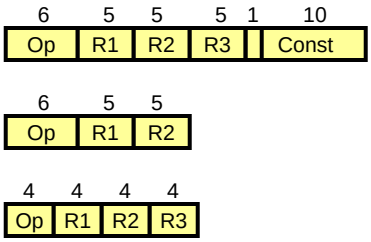
- Variable-length instructions give more efficient encodings
  - no bits to represent unused fields/operands
  - can frequency code operations, operands, and addressing modes
  - Examples
    - VAX-11, Intel x86 (byte variable)
    - Intel 432 (bit variable)
- But - can make fast implementation difficult
  - sequential determination of location of each operand



VAX instrs: 1-53 bytes!

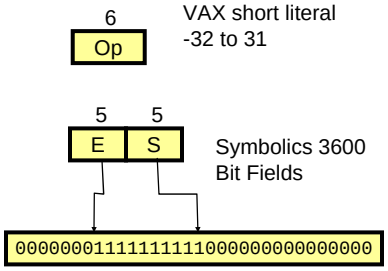
Compromise: A Few Good Formats

- Gives much better code density than fixed-format
  - important for embedded processors
- Simple to decode
- Examples:
  - ARM Thumb, MIPS 16
- Another approach
  - On-the fly instruction decompression (IBM CodePack)



Constant Encoding

- Integer constants
  - mostly small
  - positive or negative
- Bit fields
  - contiguous field of 1s within 32bits (64 bits)
- Other
  - addresses, characters, symbols
- A good architecture
  - uses a few bits to encode the most common.
  - allows any constant to be generated (table reference)

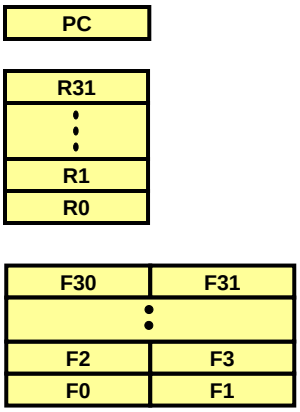


MIPS ISA

MIPS ISA

- 32 GP Integer registers (R0-31) – 32 bits each
  - R0=0, other registers governed by conventions (SP, FP, RA, etc.)
- 32 FP registers (F0-F31)
  - 16 double-precision (use adjacent 32-bit registers)
- 8, 16, and 32 bit integer data types
- Load/Store architecture (no memory operations in ALU ops)
- Simple addressing modes
  - Immediate R1 ← 0x23
  - Displacement R2 ← d(Rx) ….. 0(R3), 0x1000(R0)
- Simple fixed instruction format (3 types), 90 instructions
- Fused compare and branch
- "ISA" has pseudo instruction that are synthesized into simple sequences (ie. rotate left `rol` = combination of shift and mask)
- Designed for fast hardware (pipelining) + optimizing compilers

MIPS ISA (a visual)



R: rd ← rs1 op rs2

6	5	5	5	11
Op	RS1	RS2	RD	func

I: ld/st, rd ← rs1 op imm, branch

6	5	5	16
Op	RS1	RD	Const

J: j, jal

6	26
Op	Const

Fixed-Format

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.

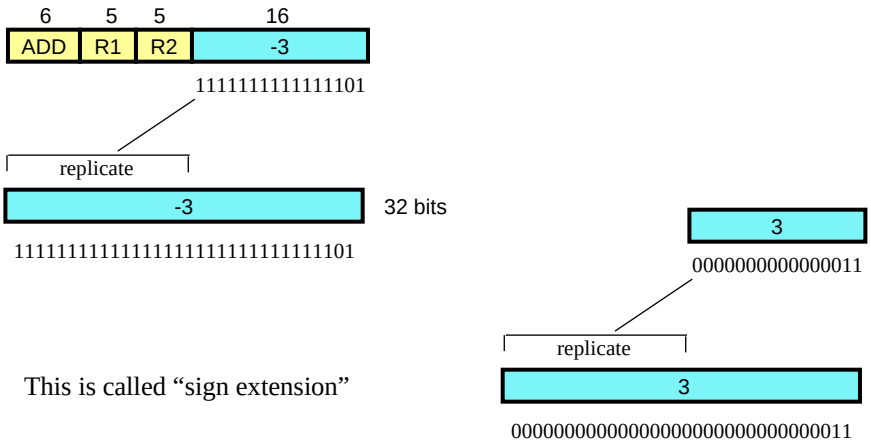
MIPS arithmetic instructions

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?

Conversion of 16 bit immediates to 32 bits

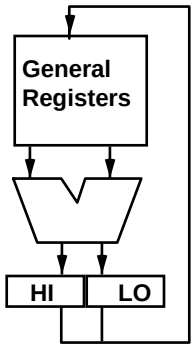
ADD R2, R1, -3



This is called “sign extension”

Multiply / Divide

- Perform integer 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



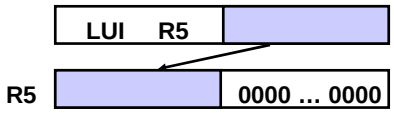
MIPS logical instructions

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 Å $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?



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

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

Summary

- ISA: memory and instructions
- MIPS as an example
  - Read more details in Appendix A
- Next Time
  - Graphics processor as another ISA example
  - ISA design principles
  - Interaction between the ISA and the compiler
- Reading assignment – Chapter 2.7 - 2.19