Lecture 3: Instruction Set Architectures

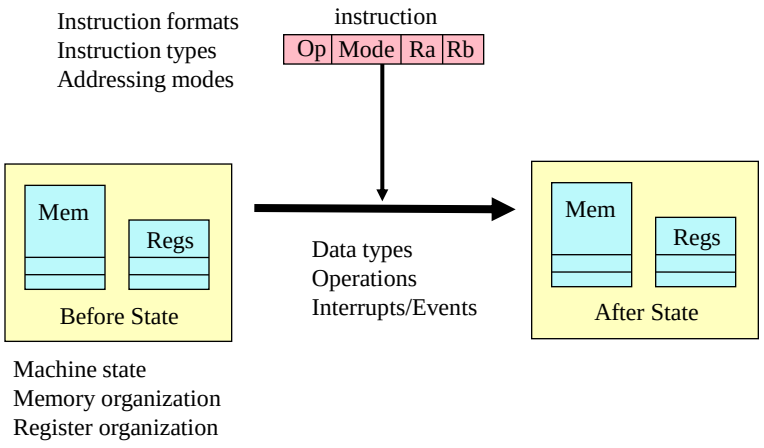
- Last Time
  - Computer elements
    - Transistors, wires, pins
- Today
  - Finish computer elements
  - ISA overview
  - MIPS ISA
  - ISA extensions
  - "Alternate" ISAs

Slides courtesy of Stephen W. Keckler, UT-Austin

Instruction Set Architecture

- Contract between programmer and the hardware
  - Defines visible state of the system
  - Defines how state changes in response to instructions
- Programmer: ISA is model of how a program will execute
- Hardware Designer: ISA is formal definition of the correct way to execute a program
- ISA specification
  - The binary encodings of the instruction set

ISA Basics



Architecture vs. Implementation

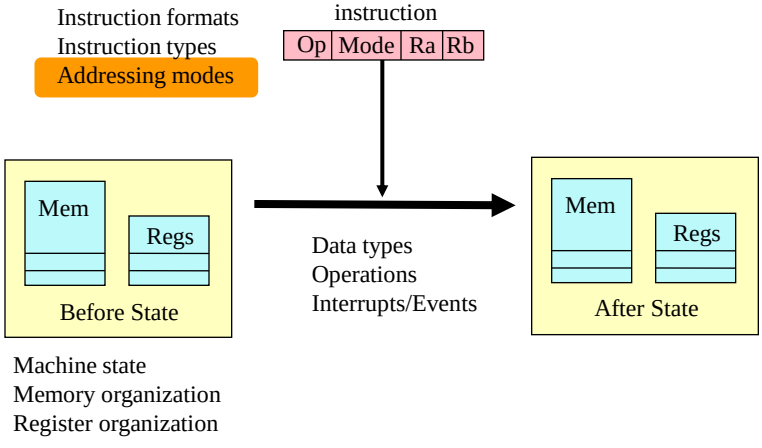
- Architecture: defines what a computer system does in response to a program and a set of data
  - Programmer visible elements of computer system
- Implementation: defines how a computer does it
  - Sequence of steps to complete operations
  - Time to execute each operation
  - Hidden "bookkeeping" functions

Classifying Instruction Set Architectures

Based on how the instructions receive/produce operands

- Stack
  - Burroughs B5000 (1963), Java Virtual Machine (late 90s)
- Accumulator
  - Univac-I (1951), EDSAC (1949)
- Register-memory
  - IBM 360 (1964), DEC PDP--and later VAX (1970), Intel x86(1978)
- Load/Store
  - IBM 801 (1974-prototype), Stanford MIPS/Berkeley RISC (mid 19080s)
  - IBM Power, MIPS, DEC Alpha, DSP, VLIW, etc.
- Dataflow
  - Numerous research machines over the years
- Special case: Vector ISAs

ISA Basics



Memory Addressing

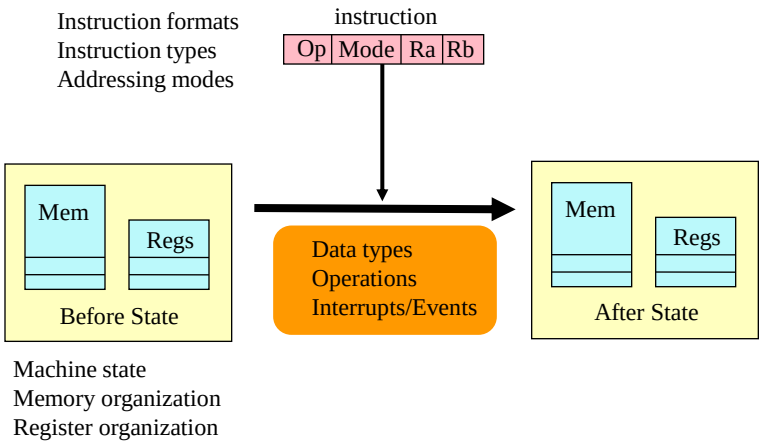
- Different size accesses (byte, half-word, word, etc.)
- Endian-ness
  - Byte ordering within a larger object
- Alignment
  - Is an access allowed to span any arbitrary boundaries?
- Addressing modes
  - How is an address computed?
  - Absolute, relative to the program counter, computed

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?)

ISA Basics



Native Data Types

- General purpose processors
  - Various size integers (now represented using 2's complement)
    - Previous included binary coded decimal, signed-magnitude, etc.
  - Floating-point (single/double precision at 32/64 bits)
  - "Addresses" - usually just integers, but not always
  - Conditions (results of comparison operations)
- But - these are not written in stone
  - Vectors (short as well as long)
  - Fixed point (often used for signal processing)
  - XYZW vertices (128 bits) for graphics

Operations (Instructions)

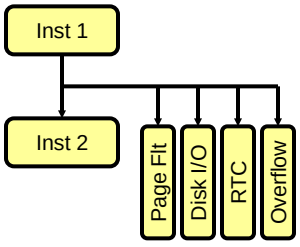
- Arithmetic/logical
- Data transfer (load/store, move)
- Control (branch, jump, call)
  - Branch delay slots
- System (operating system call, access to special state)
- Miscellaneous
  - Subword parallel, saturating arithmetic
  - Floating-point
  - Decimal (such as for BCD)
  - String operations (some found in VAX, etc.)
  - Graphics - pixel/vertex operations
- Basically - whatever you can justify and build in HW
  - But how do you decide what to put in?
    - Shouting contest
    - Democracy
    - Seniority
    - **Analysis of tradeoffs**

Instruction Encoding

- Variable length - VAX, x86
  - Instruction length depends on the number of operands, etc.
  - Intel 432 took this to an extreme
- Fixed length - MIPS, other "RISC" ISAs
  - All instructions are the same length
- Hybrid - ARM Thumb, MIPS 16
  - May support small number of fixed sizes (16/32 bits)

Control - Exceptions/Events

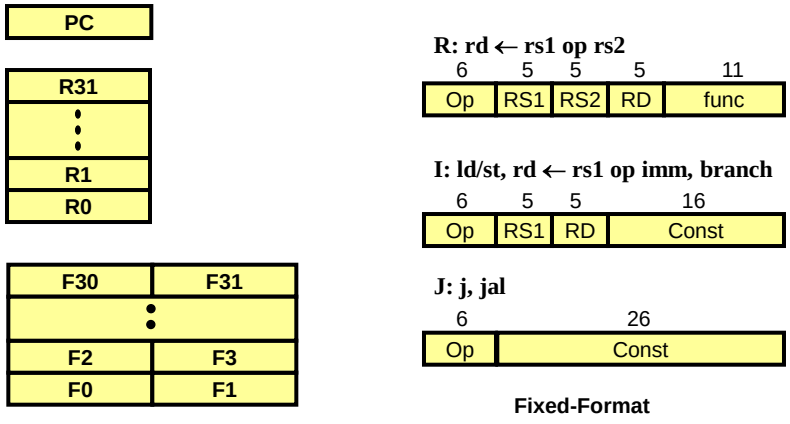
- Implied multi-way branch after every instruction
  - External events (interrupts)
    - completion of I/O operations
  - Internal events (faults or exceptions)
    - arithmetic overflow
    - page fault
- What happens????
  - EPC ← PC of instruction that caused fault
  - PC ← f(Fault type)
    - new PC from HW table lookup
  - Return: PC ← EPC + 4
- How would you use this to aid compatibility?



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)



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.

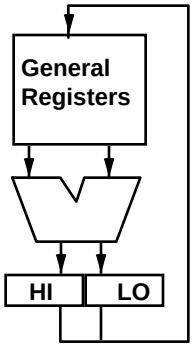
MIPS arithmetic instructions

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?

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



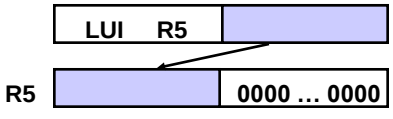
MIPS logical instructions

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	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	

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

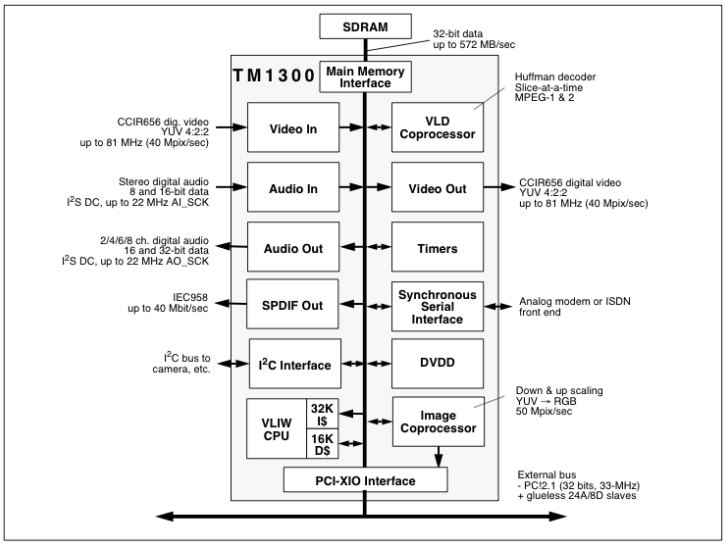
Multimedia Instruction Extensions

- Properties of multimedia applications
  - Narrower data types
    - Bytes, halfwords, fixed-point, single-precision
  - Statically known loop bounds
  - Different common idioms
    - Multiply+accumulate, averaging, permutations
- ….results in different instructions
  - Short SIMD (single instruction/multiple data)
  - Segmented arithmetic
  - Loop counters (more often in multimedia processors)
  - New instructions implemented directly in hardware

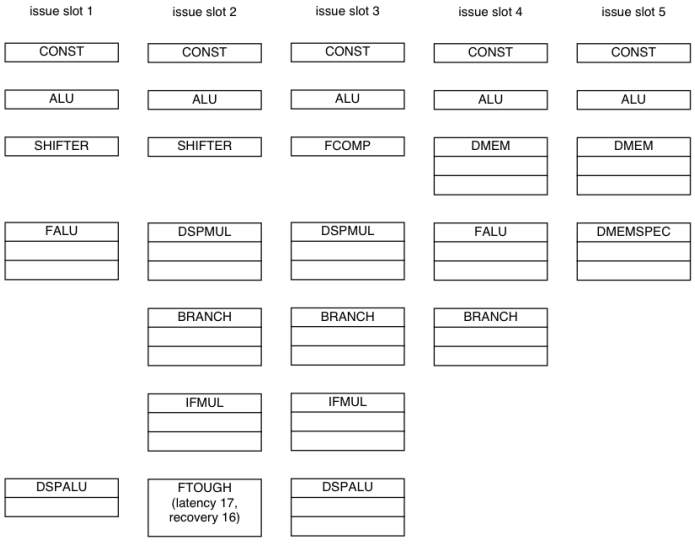
Philips Trimedia ISA (TM1300)

- VLIW instruction set
  - One instruction holds 5 independent operations
    - 27 different types of functional units
    - NOP placeholders required if operation slot is unused
  - 3 branch delay slots exposed to compiler
  - Implications
    - Hazards must be detected/prevented in SW
- Guarded/Predicated execution
  - Execution/nullification of instruction depends on value of a general purpose register
- State
  - 128 32-bit registers
  - Instructions kept compressed until delivered to processor
- Operations
  - Also uses sub-word parallelism (2 16-bit operations)
  - Saturating arithmetic

Trimedia 1300 Block Diagram



TM1300 Issue Restrictions



GPU Instruction Sets

CineFX

Vertex Processing Instruction Set

- **Add & multiply instructions**
  ADD, DP3, DP4, DPH, MAD, MOV, SUB
- **Math functions**
  ABS, COS, EX2, EXP, FLR, FRC, LG2, LOG, RCP, RSQ, SIN
- **Set on instructions**
  SEQ, SFL, SGR, SGT, SLE, SLT, SNE, STR
- **Branching instructions**
  BRA, CAL, RET
- **Address register instructions**
  ARL, ARA
- **Graphics-oriented instructions**
  DST, LIT, RCC, SSG
- **Minimum / maximum instructions**
  MAX, MIN

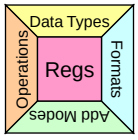
Example: DPH – homogeneous dot product

- The DPH instruction assigns the four-component dot product of the two source vectors where the W component of the first source vector is assumed to be 1.0 into the destination register.
- Semantics:

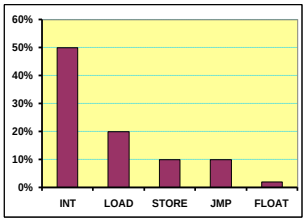
```
t.x = source0.c***;
t.y = source0.*c**;
t.z = source0.**c*;
if (negate0)  { t.x = -t.x; t.y = -t.y; t.z = -t.z; }
u.x = source1.c***;
u.y = source1.*c**;
u.z = source1.**c*;
u.w = source1.***c;
if (negate1) { u.x = -u.x; u.y = -u.y; u.z = -u.z; u.w = -u.w; }
v.x = t.x * u.x + t.y * u.y + t.z * u.z + u.w;
if (xmask) destination.x = v.x;
if (ymask) destination.y = v.x;
if (zmask) destination.z = v.x;
if (wmask) destination.w = v.x;
```

Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers

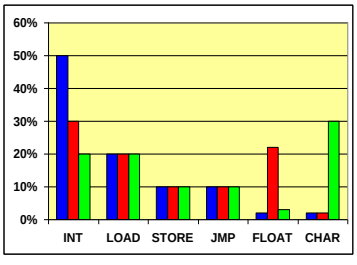


- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others

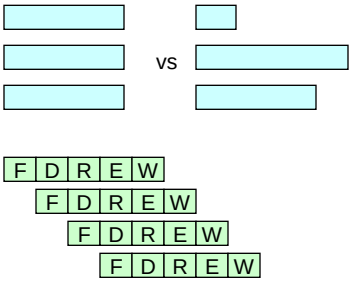


Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features or instructions
  - principle of *least surprise*
  - performance should be easy to predict



- Locality and concurrency
  - design ISA to permit efficient implementation
    - today
    - 10 years from now



Review of ISA Principles

- Good ISA design
  - KISS! - only implement necessities (encodings, address modes, etc.)
  - FOG: Frequency, Orthogonality, Generality
- Instruction Types
  - ALU ops, Data movement, Control
- Addressing modes
  - Matched to program usage (local vars, globals, arrays)
- Program Control
  - Conditional/unconditional branches and jumps
  - Where to store conditions
  - PC relative and absolute

Next Time

- Microarchitecture
  - Components + structure
- Some comments on pipelining
- Reading assignment
  - Review due:
    - Colwell, Instruction sets and beyond
    - Burger et al., Scaling to end of silicon…