Instruction Set Architecture Design

1 Introduction

- In this lecture, we are going to look at the principles and issues behind the design of instruction set architectures (ISAs).
- Then, we will explore the advantages and disadvantages of the two main ISA design philosophies: RISC and CISC.
- Finally, we will look in detail at one example ISA which we will use for the rest of the subject: the MIPS architecture.
- At the end of the journey, you will probably find it beneficial to go back over the material, to see how the MIPS architecture reflects the principles and issues of ISA design.

2 Designing an Instruction Set and CPU

- Our goal, in designing an ISA is to make a CPU which has enough functionality so that programmers can write programs for it.
- As we saw last week, we need instructions to do basic maths, data comparisons, deal with data of different sizes, and instructions to branch and jump (so we can implement decisions, loops and functions).
- There are an infinite number of ways we can build instruction sets to achieve the above, so we will also need to make some other design decisions along the way.
- However, any design decision which we make then becomes a constraint which limits all our other design decisions.
- In effect, any complete design is a collection of compromises which in total come as close as possible to achieving our original goals.
- So, we need to keep in mind what are our most important design goals, and which ones we can compromise on.

3 Basic ISA Issues

3.1 Registers

- We probably want to have some registers, so that we don't have to keep going out to slow memory, but how many registers do we want or need?
- More registers is probably better, but this will make implementing the CPU more complicated, i.e. we tradeoff ease of programming against ease of building the CPU.
    - and we need bits in each instruction to encore the register number: 8 registers = 3 bits etc.
- It also means we have to save out more registers when we switch between programs, and we do want to support a system where multiple programs can be running at the same time.
- On the other hand, too few registers means that we will have to go out to main memory more often.
- Another issue is how we deal with the special registers like the Program Counter (PC): do we want these registers visible and accessible to the programmer?
    - this would allow the programmer to change the PC's value by hand, which could be useful.

  - or do we provide special instructions to manipulate these in only limited ways?
- Most modern ISAs have 8 to 32 registers.

3.2 Bus Sizes

- What size data bus do we want? This effectively sets the natural word size of the ISA.
- And because the CPU will want to fetch in word units, this also influences the size of our instructions.
  - generally, each instruction will be 1 word at minimum, or a multiple of the word size.
- What size address bus do we want? This determines how much memory the CPU can address.
- But then we need the ability to express every address in some way.
- If the address size is too big, it can make the instruction size too big.
- For example, if the address bus is 64 bits, how will we encode the operation: load a word from address X into register R3?
  - The instruction will be at least 64 bits long to hold the address, plus bits to describe the load operation and the register which is the destination.

3.3 Operations

- How many operations do we want to have?
- If we have lots, this provides a rich set of operations that the programmer can perform, but it will make implementing them in silicon much more difficult.
- We will also need more bits in each instruction to encode the operation:
  - 5 bits => 32 operations, 6 bits => 64 operations etc.
- On the other hand, we might choose to have only a small set of simple instructions. This may force the programmer to have to combine 2 or more instructions to get something done, but the advantage is fewer bits required to encode an operation, and a simpler CPU design.

3.4 How Many Operands?

- With the number of instructions decided, how many operands will each operation require?
- If the CPU gets most of its data from registers, then we probably want to have some **3-operand** instructions like

  ADD R1, R2, R3, i.e. Add R2 and R3, and save the result into R1

  so the instruction format needs to have bits set aside to identify each of the three registers.
- Other 3-operand instructions include instructions which compare and then divert the PC to a new instruction, e.g.

  BGT R1, R2, 100, i.e. if R1 > R2, branch the CPU to instruction at current PC + 100

- Not all instructions have 3 operations. Examples of **2-operand** instructions include:

  LOAD R3, 4000, i.e. get the value from memory location 4000 and load it into register R3.

  SAVE R4, 5000, i.e. write R4's value out to memory location 5000.

  SET R6, 23, i.e. set R6 to the literal value 23 (not the value at location 23).

- And, of course, a CPU designer may think of **1-operand** instructions, such as:

  INCR R4, i.e. increment the value in R4. The Java equivalent is `R4++`.

- Later on, we will talk about the various *addressing modes* which a CPU designer might wish to use.

3.5 Literal Values

- Many instructions require literal values.
  - e.g. in Java when we write `for (i=0; i<100; i++)`, there are two literal values: 0 and 100.
- Are we going to be able to find space in each instruction to put in literal values?
- If so, that will be great, but it will be wasted space if programs don't have many literal values.
- If we can't find space, then each time there is a literal, it is going to have to be stored in a register, or we are going to have to go out to memory to fetch the literal value.

3.6 Instruction Format

- Each different instruction has to be encoded differently: operation, operands, literal values, where to place the result, size of the data being operated on etc.
- What is the instruction format going to look like?
- Can we make each instruction the same size, or are some instructions going to be different sizes?
- Is there going to be a single format, which makes the decoding in silicon easy, or are we going to have several different instruction types, each with a different format?

3.7 A Hypothetical Example

- Before we go any further, let's make the above concrete by doing some ISA design.
- Let's design an ISA with 8 registers, a 16-bit data bus and word size, and a 24-bit address bus: quite suitable for an embedded CPU, e.g. in a microwave or engine control system.
- Let's also have 3-operand instructions where the operands are all registers: $R_{dest}=R_{src1}$ OP R_{src2}
- We need 3 bits to encode each register's number, so that's 9 bits out of 16 used up, leaving 7 bits.
- Let's use 1 bit to encode the size of the data being manipulated: 8-bit byte or 16-bit int. That leaves 6 bits.
- We could go for a single instruction format:

Operation	Size	Rdest	Rsrc1	Rsrc2
6 bits	1 bit	3 bits	3 bits	3 bits

  but now there's no way to copy values between the registers and the main memory, nor is there any way to put literal values into an instruction.

- Let's add a second instruction format, still 16-bits long, but which contains a literal value:

Inst Type	Operation	Size	Rdest	Rsrc1	Rsrc2
0	5 bits	1 bit	3 bits	3 bits	3 bits

Inst Type	Operation	Size	Register	Literal Value

10	2 bits	1 bit	3 bits	8 bits

- This allows us to do things like SET R0, 26, where the 26 is stored in the 8-bit literal section.
- But we still don't have a way to access locations in memory, so there is one final instruction format:

Inst Type	Operation	Size	Register	Memory Location
11	2 bits	1 bit	3 bits	24 bits

- This instruction format is 2 words long, and the memory address is specified as a 24-bit value. This allows us to do things like LOAD R0, 4000 and SAVE R4, 5000.
- We now have:
  - a 1-word instruction format with 2^5=32 possible operations on 3 register operands,
  - a 1-word instruction format with 2^2=4 possible operations on one register and an 8-bit literal value, and
  - a 2-word instruction format with 2^2=4 possible operations on one register and a 24-bit memory location

3.8 Decision Making

- Up to now, we haven't considered how we are going to modify the value in the PC, so that it can deviate from the normal "next instruction" flow of execution.
- We need a way to skip over instructions, based on a decision, so as to implement IF statements.
- We also need to branch backward, based on a decision, so as to implement loop constructs.
- We need to jump to the start of a function, and also know how to return back to where we left.
- All of the above change the default PC behaviour: move to the address of the next instruction.
- What decisions are we going to provide? ==, !=, <, <=, >, >= ?
- Some of these we could leave out, e.g. `if (R3 < R4)` is the same as `if (R4 > R3)`.
- There are several choices here. We could design an instruction format which encodes:
  - two operands to be compared,
  - what type of comparison to make,
  - what change to make to the PC if the comparison is true.
- For example:

R3	<	R4	skip ahead 4 instructions

- An alternative is to have a **status register** with bit flags that are set on/off based on the result of the *last instruction*, and other instructions which branch depending on what bits in the status register are set.
- For example, let's have a last result was zero (Z) flag, and a last result was negative (N) flag. Other common flags hold the ALU's Carry and Overflow output.
- We can now have several branching instructions:
  - branch to instruction X if Z==1, i.e. the last instruction's result was 0.
  - branch to instruction X if Z==0, i.e. the last instruction's result was not 0.
  - branch to instruction X if N==1, i.e. the last instruction's result was negative.

  - branch to instruction X if N==0, i.e. the last instruction's result was positive.
  - branch to instruction X if N==0 and Z==0, i.e. the last instruction's result was positive and not zero, i.e. >0.
- Given the above, we can now make these comparisons just with a subtraction operation:

We Want	We Perform	The Result	Z becomes	N becomes
if R1 == R2	R1 – R2	is 0 if true	1 if true	0 if true
if R1 != R2	R1 – R2	is not 0 if true	0 if true	don't care
if R1 < R2	R1 – R2	is < 0 if true	0 if true	1 if true
if R1 <= R2	**R2 – R1**	is >= 0 if true	don't care	0 if true
if R1 > R2	R1 – R2	is > 0 if true	0 if true	0 if true
if R1 >= R2	R1 – R2	is >= 0 if true	don't care	0 if true

- Let's now turn our attention to two of the most popular ISA design philosophies: RISC and CISC.
- They are nearly diametrically opposed in term of how to achieve their goals, but the goal is the same: to design an ISA which which has enough functionality so that programmers can write programs for it.

4 CISC: Complex Instruction Set Computing

- In the early silicon days of computing, there were a lot of constraints on ISA design:
  - memory was expensive.
  - CPUs were made from few discrete transistors, or with chips that had few integrated transistors.
- For these reasons, designers wanted ISAs which could pack instructions into as few bits as possible; similarly, CPUs could not afford to have too many registers due to the limited transistor numbers.
- At the same time, most programmers were still using assemblers, i.e. down close to the instruction set, or using the early compilers which had to translate down to the instructions available in the instruction set.
- There was also a mistrust in the efficiency of the compilation process, both in terms of program speed and efficient use of the memory.
- With all of the above pressures, ISA designers created instruction sets with instructions to do everything, and the instructions themselves performed complex operations. Instruction sets of this type are now known as **CISC** architectures.
- CISC ISAs made writing programs in assembly much easier for the programmers, and at the same time, they made translating high-level abstractions such as loops and function calls easier, thus aiding the compiler writers.
- With fewer registers, there were more memory accesses, and the CISC instructions also provided many complex addressing modes to access memory in ways that resembled high-level constructs such as pointers, single- and multi-dimensional arrays, structures and objects, linked lists etc.
- To support all these addressing modes, and the operands needed to specify them, CISC instructions are often variable in length.
- As a result, the number of memory accesses to perform a CISC instruction can vary from 1 (instruction fetch) to several, e.g. instruction fetch, address of operand fetch, fetch of value at that address, use that value as a pointer to fetch another value, ditto for the second operand, perform the operation, write the result back to memory.

- This implies that CISC instructions take variable amounts of time to execute, depending on their complexity and the number of memory accesses required.
- The VAX ISA, designed in the late 1970s, is considered to be the epitome of the CISC philosophy.
- It has 16 32-bit registers, 11 different data types, 15 different addressing modes, instructions whose sizes range from 1 byte up to around 60 bytes, and such instructions as:
  - LOCC: locate character in character string
  - CRC: calculate cyclic redundancy check
  - CVTPL: convert packed decimal string to longword
  - INSQUEUE: insert element into queue
- For a good comparison of the VAX and MIPS ISAs, read this appendix from Patterson and Hennessy.
- The Intel IA-32 ISA, used on the 80386, 80486, Pentium CPUs and their descendants, is also a strongly CISC-oriented architecture.
- IA-32 has only 4 general-purpose registers, several special-purpose registers including a PC, a status register, and several memory-management registers.
- Instructions are variable-length, from 1 to several bytes. There are several addressing modes and data types, but not as many as the VAX.
- IA-32 provides hundreds of instructions, and new ones keep being added as the architecture evolves (e.g. MMX, SSE, SSE2).

5 RISC: Reduced Instruction Set Computing

- In the 1970s, as memory sizes and transistor counts improved, there was a backlash against CISC.
- Several researchers demonstrated that some of the more complex CISC instructions actually took longer to execute than a sequence of simpler instructions.
- Moreover, many of the esoteric instructions and addressing modes were simply not being used by programmers or by compilers. It was too hard, for example, to automate the translation of high-level constructs to make use of all the features of a CISC architecture.
- With more CPU transistors, the need to limit the number of registers was removed, which also reduced the frequency that the CPU needed to access main memory.
- Another researcher also noted that 98% of all literal values used in programs could be represented in 13 bits or less, yet these were being stored as 16-bit or 32-bit values in CISC designs.
- All of the above drove a shift in design towards ISA architectures where the instructions were simpler, there were more registers, the number of addressing modes was reduced, and instruction formats were simplified to be only 1 word long: Reduced Instruction Set Computing, or **RISC**.
- The word "Reduced" here is a misnomer. It does not mean that the number of operations was reduced (although this did happen), but that the instructions available performed simpler tasks than CISC counterparts, e.g. there were no instructions like the VAX INSQUEUE instruction.
- With a simpler set of instructions, and a simplified instruction format, the transistors available on the chips could be taken away from the decode section and used to improve CPU performance:
  - pipelining was introduced, where several instructions could be passing through the CPU at the same time, at various stages of decode and execution
  - there were more transistors to create on-CPU data and instruction caches
  - the clock speed of CPUs could be increased, as a result of the simplified operations and instruction formats.

- Typical RISC characteristics include (from Wikipedia):
  - Uniform instruction format, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding;
  - Identical general purpose registers, allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers);
  - Simple addressing modes. Complex addressing is performed via sequences of arithmetic and/or load–store operations;
  - Few data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is unlikely to be found on a RISC.
- While Intel's IA–32 architecture has become the dominant ISA design on desktops and laptops, RISC CPUs have also been popular, and carved out a niche in areas such as embedded devices, games consoles and low–power environments such as mobile phones.
- Examples of RISC CPUs include:
  - ARM, which is used on such devices as Apple's iPhones, iPods and iPads, and most Android phones.
  - PowerPC, which is used on the Nintendo Wii, the Gamecube and network devices such as switches and routers.
  - Freescale's Power Architecture, used in the Xbox 360 and Playstation 3.
  - the Atmel AVR, used in embedded environments such as motor cars.
- For the rest of this subject, we will be using a RISC architecture, the MIPS ISA, as our representative architecture both for studying instruction sets and for examining how a CPU actually works.
- MIPS was the pioneer chip–based RISC architecture, originally designed in 1981 at Stanford University, and one of the first CPUs to be heavily pipelined to improve performance.
- The first MIPS CPU, the R2000, was released in 1985, and several other members of the ISA family have been released since then.
- Now sidelined a bit by the recent success of the ARM ISA, MIPS has been used in such devices as the the PlayStation, PlayStation 2, Nintendo 64, PlayStation Portable, and embedded devices & home routers like the Linksys WRT54G series.

6 The MIPS Instruction Set Architecture

- The MIPS architecture is the epitome of the RISC philosophy.
- The data bus size (and the word size) is 32 bits, and the address bus is 32 bits too.
- MIPS supports 4 integer data types: 8-bit bytes, 16-bit halfwords, 32-bit words and 64-bit doublewords.
- Floating point types are also supported: 32-bit single-precision and 64-bit double-precision values.
- MIPS provides 32 general-purpose 32-bit registers, although the operating system, assembler and libraries reserve some of these for special use.
- MIPS is an example of a **load/store architecture**: data has to be loaded from memory into registers first before it can be manipulated. Results are stored into registers, and they have to be stored out to memory if required.
- MIPS provides instructions to perform the following operations:
  - load registers with values, either from RAM or with literal values.
  - store register values (i.e. copy them) out to RAM locations.
  - basic integer arithmetic: add, subtract, multiply, divide with remainder.
  - basic floating-point arithmetic: add, subtract, multiply, divide.
  - logical operations: AND, OR, NOR, exclusive OR (XOR).
  - shift operations: shift left, shift right.
  - comparison operations: ==, !=, <, >, <=, >=
  - instructions to change the flow of control: relative branches and jumps.
- There are only 3 instruction formats, all of which are one word (32 bits) in size:
  - R-type instructions, which perform arithmetic and logical operations on registers.
  - I-type instructions, which deal with load/stores and immediate literal values, as well as branches.

- J-type instructions, which are used for jumps and function calls.

- All three formats share the same 6-bit operation field, so different values here determine what type each instruction is.

6.1 R-type Instructions

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- R-type instructions perform arithmetic and logical operations on registers.
- The R-type instruction format has 3 operands and six fields in total:
  - *op*: the operation of the instruction: 6 bits provides for up to 64 operations
  - *rs*: the first register operand: 5 bits encodes all 32 register values
  - *rt*: the second register operand
  - *rd*: the destination register, which gets the result
  - *shamt*: the shift amount, ignore this for now
  - *func*: function; this field modifies the operation specified in the op field
- Examples of R-type MIPS instructions are shown below.

Instruction	op	rt	rt	rd	shamt	func
add $1, $2, $3	0	2	3	1	0	32
addu $1, $2, $3	0	2	3	1	0	33
subu $1, $2, $3	0	2	3	1	0	35
mfc0 $1, $epc	16	0	1	14	0	0
mult $2, $3	0	2	3	0	0	24
multu $2, $3	0	2	3	0	0	25
div $2, $3	0	2	3	0	0	26
and $1, $2, $3	0	2	3	1	0	36
or $1, $2, $3	0	2	3	1	0	37

- Note that the *op* is often 0 whereas the *func* changes: perhaps it's better to say that the *func* specifies the operation and the *op* modifies it!
- MIPS has several special-purpose registers:
  - HI and LO are 32-bit registers which are the destination for multiplies and divides, which is why the mult and div instructions only seem to have 2 operands: HI & LO and the implicit destination.
  - other special-purpose registers like $epc deal with interrupts and exceptions, so the *mfc0* instruction allows the programmer to access their values with a move to a general-purpose register.

6.2 I-type Instructions

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

- I-type instructions deal with load/stores and immediate literal values.
- The I-type instruction format has 3 operands and four fields:
  - *op*: the operation of the instruction
  - *rs*: the first register operand: 5 bits encodes all 32 register values
  - *rt*: the second register operand
  - *address/immediate*: the third operand, either a literal value or 16-bits of a 32-bit address
- Note that the immediate field follows the research that 98% of all literal values used in programs could be represented in 13 bits or less.
- I-type instructions can be used to perform arithmetic where one of the operands is a literal value, e.g.

Instruction	op	rs	rt	immed
addi $1, $2, 64	4	2	1	100

- They can also be used to load 8-bit or 16-bit literals into registers.

Instruction	op	rs	rt	immed
lh $1, 400	33	0	1	400

- To load larger values into registers, two I-type instructions are required, and the assembler will translate a *pseudo-instruction* into multiple instructions.
- For example, to load the 32-bit literal value 0x12345678 into register $1, we can write `lw $1, 0x12345678`. The assembler translates this into two instructions:
  - the *lui* (load upper immediate) instruction loads 0x1234 into the top 16-bits of $1, setting the bottom 16-bits to zeroes,
  - then the *ori* (OR immediate) instruction loads the remaining 0x5678 into the bottom 16-bits of $1.
- Similarly, instructions to load or store values to/from memory require two instructions, as memory addresses are 32-bits whereas the address field in the I-type instruction is only 16-bits:
  - the first instruction loads half of the address into a register.
  - the second instruction combines the first half in the register with the second half in the address field, accesses the values at the full 32-bit memory address, and stores it in the remaining register operand.
- Fortunately, with 32 registers, the need to access main memory is less severe, and this reduces the need for instruction pairs.

6.3 I-type Branches

- Because I-type instructions hold a literal value, they can also be used to do branches (where the PC change is relative).
- Consider the following high-level language code fragment:

```
if (i == j)
  f = g + h;
f = f - i;
```

- Assume that MIPS has stored *f*, *g*, *h*, *i* and *j* in registers $16 through to $20.

- This can be translated to the MIPS instructions:

```
     bne $19, $20, L1        # skip if (i != j), note the not equals test
     add $16, $17, $18       # f = g + h
L1:  sub $16, $16, $19       # f = f - i
```

- The branch instructions add or subtract from the address of the next instruction, i.e. modify the PC, so as to skip past instructions or to loop back to repeat instructions.

Instruction	op	rs	rt	immed
bne $19, $20, 2	5	19	20	2

- Because all instructions are 1 word or 4 bytes long, the immediate value is multiplied by 4 before it is added to the PC.
- So, the 16-bit immediate value can jump forwards or backwards 32,767 instructions (not 32,767 bytes).

6.4 J-type Instructions

op	target address
6 bits	26 bits

- J-type instructions are used for jumps (where the PC change is absolute) and function calls.
- The 26-bit target address is multiplied by 4, and the new value is stored in the program counter.
- This makes the jump value 28-bits, still not the full 32-bits available for full addresses.
- To jump more than 28-bits of distance, you need to chain multiple jump instructions, each around 28-bits of address apart.
- We will explore the branch and jump instructions in more detail later, as well as the MIPS addressing modes.
- For now, here is a table from Patterson and Hennessy describing the basic MIPS instructions, grouped according to function (not instruction type).

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	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
	Move fr. copr. reg.	mfc0 $1,$epc	$1 = $epc	Used to get exception PC
	multiply	mult $2,$3	Hi, Lo = $2 ¥ $3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $2,$3	Hi, Lo = $2 ¥ $3	64-bit unsigned product in Hi, Lo
	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 and remainder
	Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
	Move from Lo	mflo $1	$1 = Lo	Use to get copy of Lo
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; logical OR
	and immediate	andi $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	ori $1,$2,100	$1 = $2 | 100	Logical OR register, 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
Data transfer	load word	lw $1,100($2)	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,100($2)	Memory[$2+100] = $1	Data from register to memory
	load upper imm.	lui $1,100	$1 = 100 x 2^{16}	Loads constant in upper 16 bits
Conditional branch	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 complement
	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 number
	set l.t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0	Compare < constant; natural
Unconditional jump	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

- For more information, see the cheat sheets and tutorials on the subject's resource page.

File translated from TEX by TTH, version 3.85.
On 25 Jan 2012, 18:37.