2017-09-19

- The words of a computer's language are called **instructions**, and its vocabulary is called an **instruction set**.
- **Stored-program concept**: The idea that instructions and data of many types can be stored in memory as *numbers*, leading to the stored-program computer.
- MIPS instruction set architecture:
  - Operands: registers, memory words.
  - Operators: arithmetic, data transfer, logical, conditional branch, unconditional jump.
- Design principles:
  - Simplicity favors regularity.
  - Smaller is faster.
  - Good design demands good compromises.
- **Word**: The natural unit of access in a computer; corresponds to the size of a register in the MIPS architecture.
- **Registers**:

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

- It is the compiler's job to associate program variables with registers.
- **Memory words**:
  - To access a word in memory, the instruction must supply the memory *address*.
  - Expression in assembly language: offset($base register).
- **Data transfer instructions**: A command that moves data between memory and registers.
  - *Load* values from memory into registers.
  - *Store* result from register to memory.
- **Alignment restriction**: A requirement that data be aligned in memory on natural boundaries,

e.g. multiples of 4 bytes in MIPS.

- The process of putting less commonly used variables (or those needed later) into memory is called **spilling** registers.
- **Constants** or **immediate operands**:
  - Constant data specified in an instruction.
  - Immediate operand avoids a load instruction.
- **Endianness**:
  - **Big-endian**: most-significant bit at least address of a word.
  - **Little-endian**: least-significant bit at least address of a word.
- Binary numbers:
  - **Two's complement**: $x + \bar{x} = -1$
  - **Sign extension**: replicate the sign bit to the left.
- Instructions are kept in the computer as a series of high and low electronic signals and may be represented as *numbers*.
- **Machine language**: the numeric version of instructions.
- The instruction formats are distinguished by the values in the first field (op).
- The desire to keep all instructions the same size is in conflict with the desire to have as many registers as possible.
- Most instruction sets today have 16 or 32 general purpose registers.
- **Binary compatibility** often leads industry to align around a small number of instruction set architectures.
- Instruction fields:
  - *op*: **operation code (opcode)**.
  - *rs*: first source register number.
  - *rt*: second source register number.
  - *rd*: register destination operand.
  - *shamt*: shift amount.
  - *funct*: **function code**.
  - *constant* or *address*.
- **Instruction format**: a form of representation of an instruction composed of fields of binary numbers.

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	address/immediate (16)			Load/store, branch, immediate
J-format	op	target address (26)					Jump

- Operators and operations:

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
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 on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional 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

- **Basic block**: a sequence of instructions without branches (except possibly at the end), and without branch targets or branch labels (except possibly at the beginning).
- MIPS compilers use the slt, slti, beq, bne, and the fixed value of $zero to create all relative conditions, and therefore, avoid higher cost of more complicated branch on less than.
  - `if (s0 <= s1) else : slt $t0, $s1, $s0 ; bne $t0, $zero, else`
  - `if (s0 < s1) else : slt $t0, $s0, $s1 ; beq $t0, $zero, else`
- Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays.
- Implementing *switch* is via a sequence of conditional tests, or via a **jump address table** (or **jump table**), a table of addresses of alternative instruction sequences.
- **Jump register (jr)** instruction is an unconditional jump to the address specified in a register.
- **Procedure** (or **function**): a stored subroutine that performs a specific task based on the parameters with which it is provided.
- **Program counter (PC)** is the register that contains the address of the instruction in the program being executed.

- **Jump-and-link (jal)** instruction jumps to an address and simultaneously saves the address of the following instruction in register $ra.
- The **jump register (jr)** jumps to the address stored in register $ra.
- The ideal data structure for spilling registers is a *stack*, which "grow" from higher addresses to lower addresses.
- The **stack pointer ($sp)** is adjusted by one word for each register that is saved or restored.
- **Temporary registers ($t0–$t9)** are not preserved by the callee (called procedure) on a procedure call.
- **Saved registers ($s0–$s7)** must be preserved on a procedure call (if used, the callee saves and restores them).
- One solution to nested procedures is to push all the other registers that must be preserved onto the stack.
- The stack is also used to store variables that are local to the procedure but do not fit in registers, such as local arrays or structures.
- **Procedure frame** (or **activation record**): the segment of the stack containing a procedure's saved registers and local variables.
- **Frame pointer ($fp)** points to the first word of the frame of a procedure.
- A frame pointer offers a stable base register within a procedure for local memory-references.
- The stack is adjusted only on entry and exit of the procedure.
- The MIPS memory allocation for program and data (from low to high): reserved, text segment (machine code), static data segment, heap, stack.
- Three choices for representing a string:
    - The first position of the string is reserved to give the length of a string.
    - An accompanying variable has the length of the string (as in a structure).
    - The last position of a string is indicated by a character used to mark the end of a string.
- MIPS' instructions to move bytes and *halfwords*: **load byte (lb)**, **store byte (sb)**, **load half (lh)**, **store half (sh)**.
- The MIPS instruction set includes the **instruction load upper immediate (lui)** specifically to set the upper 16 bits of a constant in a register.
    - Load the upper 16 bits using *lui*.
    - Insert the lower 16 bits using *ori*.
- Either the compiler or the assembler must break large constants into pieces and then reassemble them into a register.
- The size of address is 16 bits in I-format or 26 bits in J-format.
- **Addressing modes**: addressing mode One of several addressing regimes delimited by their varied use of operands and/or addresses.
    - **Immediate addressing**: the operand is a constant within the instruction itself.

    - **Register addressing**: the operand is a register.
    - **Base or displacement addressing**: the operand is at the memory location whose address is the sum of a register and a constant in the instruction.
    - **PC-relative addressing**: the branch address is the sum of the PC and a constant in the instruction, i.e. *branch address* = *program counter* + *register*.
    - **Pseudodirect addressing**: the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC.
- Conditional branches are found in loops and in if statements, so they tend to branch to a nearby instruction.
- If a branch target is too far away to be encoded with 16-bit offset, assembler rewrites the code and makes use of jump (j) instruction.