Machine Instructions

- The machine can only understand *instructions*.
- The *instruction set* is the vocabulary of the machine.
- The *instruction set architecture* (ISA) defines the interface between software and hardware.
- The ISA allows different machine implementations to run the same software.
MIPS Add/Sub:

- **C program:**
  - `a = b + c;`
  - `d = a - e;`

- **MIPS instructions:**
  - `add a, b, c`
  - `sub d, a, e`

- Always three operands:
  - A fixed number of operands makes the hardware simpler.
C compiler example

- **C expression:**
  - $f = (g + h) - (i + j)$;

- **C compiler generates these instructions:**
  - `add $t0, g, h`
  - `add $t1, i, j`
  - `sub f, $t0, $t1`

- **Temporary variables:** $t0$, $t1$
Registers

- Instruction operands are in *registers*
- Temporary variables $t0$, $t1$ are stored in registers.
  - What about f,g,h,i,j ?
- The MIPS ISA defines “only” 32 registers!
  - Each register 32 bits = *word size* of the architecture.
  - C integer (*int*) typically matches word size.
C compiler example with registers

- C expression:
  - $f = (g + h) - (i + j)$;

- C compiler register assignment:
  - $f, g, h, i, j$ are assigned to: $s_0, s_1, s_2, s_3, s_4$
  - $t_0, t_1$: $t_0, t_1$

- Instructions:
  - add $t_0, s_1, s_2$
  - add $t_1, s_3, s_4$
  - sub $s_0, t_0, t_1$

- The 32 registers have special names (Fig. 3.13)
# Register Names

<table>
<thead>
<tr>
<th>Name</th>
<th>Reg #</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Results and expressions</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>Temp</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>“saved”</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>Temp</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global Pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return Address</td>
</tr>
</tbody>
</table>
Accessing Memory

- Arithmetic instructions can only access registers.
- We need to transfer data between memory and registers.
- Transferring from memory to register:
  - The *Load Word* (lw) instruction can move 32 bits of data from a memory address to a register.
  - The *Store Word* (sw) instruction can move 32 bits of data from a register to a memory address.
- A 32-bit register can store an address.
C compiler example with load word

- C expression:
  - `int A[100];  // int is 4 bytes.`
  - `g = h + A[8];`

- The **base address** of array A is in register $s3

- **Offset** into array is 8 words or 32 bytes as the MIPS uses byte addressing.
  - One 32-bit word = 4 bytes.
  - Words MUST be 4 byte aligned.

- C compiler generates a lw-instruction:
  - `lw $t0, 32($s3)`
  - `add $s1, $s2, $t0`
32-bit word alignment in byte addressed memory

- Aligned at 4 (Good):

```
<table>
<thead>
<tr>
<th>4</th>
<th>MSB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Unaligned at 5 (Bad!):

```
<table>
<thead>
<tr>
<th>5</th>
<th>LSB</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
C compiler example with load/store

- **Example 1**
  - C expression:
    - $A[12] = h + A[8]$; ($A$ is in $s_3$, $h$ is in $s_2$)
  - C compiler generates lw- and sw-instructions:
    - lw $t0, 32(s_3)$
    - add $t0, s_2, t0$
    - sw $t0, 48(s_3)$

- **Example 2**
  - Variable array index: load from $A[i]$ ($i$ in $s_4$)
    - add $t1,s_4,s_4$ # $2*i$
    - add $t1,t1,t1$ # $4*i$
    - add $t1,t1,s_3$ # $A + 4*i$
    - lw $t0, 0(t1)$
Representing Instructions (Arithmetic)

- **C code:**
  - `temp = (g + h);`

- **MIPS assembly instruction:**
  - `add $t0,$s1,$s2 # add rd,rs,rt : rd=rs+rt`

- **Machine language instruction:**
  - `000000 10001 10010 01000 00000 100000`
  - R-type (register) instruction format (32 bits):

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
Representing Instructions (Memory)

- C code:
  - `temp = A[8];` (A is an int array)

- MIPS assembly instruction:
  - `lw $t0, 32($s3)` # lw rt,offset(rs): rt=mem[rs+offset]

- Machine language instruction:
  - `100011 10011 01000 0000000000100000`
  - I-type instruction format (32 bits):

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate value = address offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- Note: the immediate field is coded in 2’s complement
The Stored-Program Concept

- Machine Instructions are represented as 32 bit numbers.
- Programs are stored in memory as a sequence of instructions.
- The processor executes instructions in sequence:
  - The Program Counter (PC) contains the memory address of the next instruction to execute.
  - For each instruction PC is incremented by 4.
The Program Counter

PC = 8

Instruction Memory

- Instruction
- Next instruction at PC+4

0
4
8
12
16
Conditional Branches and Unconditional Jumps

- C code:

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

```
if (i != j)  Go to Label_1
T
F
+  -
```

```
add     $s0,$s1,$s2
Label_2:
```
Conditional Branch instructions

- Branches can test for equality:
  - `bne a,b,Label`  # go to Label if a != b
  - `beq a,b,Label`  # go to Label if a == b

- But cannot test for a<b, etc.
  - Must be done using other instructions.

- The `slt` R-type instruction is used for a<b tests:
  - `slt $t0,a,b`  # $t0 = 1 if a<b else 0
  - `bne $t0,$zero, Label`  # go to Label if $t0 != 0 (a<b)
    - Register 0 ($zero) always contains a zero.
Representing Instructions (Branch)

- **code:**
  - if (a == b) go to LABEL;

- **MIPS assembly instruction:**
  - [0x0000009C] beq $s1,$s2,label  # if $s1==$s2 PC+=4+(25*4)
  - [0x000000A0] nop
  - ...
  - [0x00000104] label:

- **Machine language instruction:**
  - 000100 10001 10010 0000000000011001 (25)
  - I-type instruction format (32 bits):
    - op 6 bits
    - rs 5 bits
    - rt 5 bits
    - immediate value = address offset 16 bits
Representing Instructions (Jumps)

- code:
  - go to LABEL;

- MIPS assembly instruction:
  - j 10000 # PC = PC[31:28] & (2500*4)

- Machine language instruction:
  - 000010 0000000000000000100111000100 (2500)
  - J-type instruction format (32 bits):

<table>
<thead>
<tr>
<th>op</th>
<th>jump target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

New target address: 4 bits (PC) + 26 bits (Instruction) + 2 bits (’00’)
Other Jump Instructions

- **Jump (J-type):**
  - j 10000  # jump to address 10000

- **Jump Register (NB: R-type!):**
  - jr rs  # jump to 32 bit address in register rs

- **Jump and Link (J-type):**
  - jal 10000  # jump to 10000 and save PC in R31
  - Use jal for procedure calls, it saves the return address (PC+4) in register 31 ($ra)
  - Use “jr $ra” to return from subroutine
Immediate data instructions (I-type)

- **Add Immediate:**
  - addi $sp,$sp,4  
    # add rt,rs,const : rt=rs+const
  - NB: The constant is sign extended to 32 bits first!

- **Load Upper Immediate:**
  - lui $s0,61  
    # rt = 61 << 16 (low bits zeroed)
  - Can be used together with e.g. addi, ori to produce 32 bit constants.

- **Load Immediate is *not* a machine instruction!**
  - The assembler has “li”, but it’s a pseudo-instruction that is expanded into lui and ori.
Summary of MIPS addressing modes

- Register addressing
  - Used by R-type instructions (add, sub)
- Immediate addressing
  - Used by immediate I-type instructions (addi, ori)
- Base addressing
  - Used by I-type instructions (lw, sw, lb, sb)
- PC-relative addressing
  - Used by branching I-type instructions (beq, bne)
- Pseudodirect addressing
  - Used by J-type jump instructions (j, jal)
Summary of MIPS instruction formats

- **R-type (add, sub, slt, jr)**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

- **I-type (beq, bne + addi, lui + lw, sw)**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate value / address offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **J-type (j, jal)**

<table>
<thead>
<tr>
<th>op</th>
<th>jump target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>
Running a C program

C source file + headers → C compiler → Assembly file → Assembler → Object code file → Other object files → Static library files

Dynamic library files → Loader (OS) → Machine language program in memory

Executable file → RUN!

Exports

.s
.o
.exe
MIPS memory map convention

Note: Hex addresses

$sp=7ffffffc  →  7ffffffc

$gp=10008000  →  10010000
10000000

PC=00400000  →  00400000
00000000

Stack

Dynamic data (heap)

Static data

Reserved (for OS)

Text (instruction memory)