Instruction Set
Architecture

Hung-Wei Tseng
Setup your i-clicker

- Register your i-clicker
  - Read here: https://csemoodle.ucsd.edu/mod/resource/view.php?id=12303

- Set your channel to “CA”
  - Press on/off button for 2 seconds
  - Press C and then press A
How we talk to computers
In the very old days...

- Physical configuration specified the computation a computer performed

The difference engine

ENIAC
The stored program computer

- The program is data
  - a series of bits
    - these bits are “instructions”!
  - lives in memory
- Program counter
  - points to the current instruction
  - processor “fetches” instructions from where PC points.
  - advances/changes after instruction execution

Processor

PC

instruction memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Operation</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>120007a30:</td>
<td>0f00bb27</td>
<td>ldah gp,15(t12)</td>
<td>120007a34:</td>
</tr>
<tr>
<td>120007a34:</td>
<td>509cbd23</td>
<td>lda gp,-25520(gp)</td>
<td>120007a38:</td>
</tr>
<tr>
<td>120007a38:</td>
<td>00005d24</td>
<td>ldah t1,0(gp)</td>
<td></td>
</tr>
<tr>
<td>120007a3c:</td>
<td>0000bd24</td>
<td>ldah t4,0(gp)</td>
<td></td>
</tr>
<tr>
<td>120007a40:</td>
<td>2ca422a0</td>
<td>ldlt0,-23508(t1)</td>
<td></td>
</tr>
<tr>
<td>120007a44:</td>
<td>130020e4</td>
<td>beq t0,120007a94</td>
<td></td>
</tr>
<tr>
<td>120007a48:</td>
<td>00003d24</td>
<td>ldah t0,0(gp)</td>
<td></td>
</tr>
<tr>
<td>120007a4c:</td>
<td>2ca4e2b3</td>
<td>stl zero,-23508(t1)</td>
<td></td>
</tr>
<tr>
<td>120007a50:</td>
<td>0004ff47</td>
<td>clr v0</td>
<td></td>
</tr>
<tr>
<td>120007a54:</td>
<td>28a4e5b3</td>
<td>stl zero,-23512(t4)</td>
<td></td>
</tr>
<tr>
<td>120007a58:</td>
<td>20a421a4</td>
<td>ldq t0,-23520(t0)</td>
<td></td>
</tr>
<tr>
<td>120007a5c:</td>
<td>0e0020e4</td>
<td>beq t0,120007a98</td>
<td></td>
</tr>
<tr>
<td>120007a60:</td>
<td>0204e147</td>
<td>mov t0,t1</td>
<td></td>
</tr>
<tr>
<td>120007a64:</td>
<td>0304ff47</td>
<td>clr t2</td>
<td></td>
</tr>
<tr>
<td>120007a68:</td>
<td>0500e0c3</td>
<td>br 120007a80</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Set Architecture (ISA)

• The contract between the hardware and software
• Defines the set of operations that a computer/processor can execute
• Programs are combinations of these instructions
  • Abstraction to programmers/compilers
• The hardware implements these instructions in any way it choose.
  • Directly in hardware circuit
  • Software virtual machine
  • Simulator
  • Trained monkey with pen and paper
From C to Assembly

- C program
  - Compiler
    - Assembly
      - Assembler
      - Object
        - Linker
          - Executable
            - Loader
              - Memory
              - Machine code/binary
              - Library
Example ISAs

- **x86**: intel Xeon, intel Core i7/i5/i3, intel atom, AMD Athlon/Opteron, AMD FX, AMD A-series
- **MIPS**: Sony/Toshiba Emotion Engine, MIPS R-4000(PSP)
- **ARM**: Apple A-Series, Qualcomm Snapdragon, TI OMAP, nVidia Tegra
- **DEC Alpha**: 21064, 21164, 21264
- **PowerPC**: Motorola PowerPC G4, Power 6
- **IA-64**: Itanium
- **SPARC** and many more ...
ISA design
What ISA includes?

- **Instructions:** what programmers want processors to do?
  - Math: add, subtract, multiply, divide, bitwise operations
  - Control: if, jump, function call
  - Data access: load and store

- **Architectural states:** the current execution result of a program
  - Registers: a few named data storage that instructions can work on
  - Memory: a much larger data storage array that is available for storing data
  - PC: the number/address of the current instruction
What should an instruction look like?

- **Operations**
  - What operations?
  - How many operations?
- **Operands**
  - How many operands?
  - What type of operands?
    - Memory/register/label/number (immediate value)
- **Format**
  - Length
  - Formats?

```
y = a + b
```

```
add r1, r2, r3
add r1, r2, 64
```
We will study two ISAs

• MIPS
  • Simple, elegant, easy to implement
    • That’s why we want to implement it in CSE141L
  • Designed with many-year ISA design experience
  • The prototype of a lot of modern ISAs
    • MIPS itself is not widely used, though

• x86
  • Ugly, messy, inelegant, hard to implement, ...
  • Designed for 1970s technology
  • The dominant ISA in modern computer systems

You should know how to write MIPS code after this class

You should know how to read x86 code after this class
MIPS ISA

- All instructions are 32 bits
- 32 32-bit registers
  - All registers are the same
  - $zero is always 0
- 50 opcodes
- 3 instruction formats
  - R-type: all operands are registers
  - I-type: one of the operands is an immediate value
  - J-type: non-conditional, non-relative branches

<table>
<thead>
<tr>
<th>name</th>
<th>number</th>
<th>usage</th>
<th>saved?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>N/A</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>assembler temporary</td>
<td>no</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS ISA (cont.)

- Only load and store instructions can access memory
- Memory is “byte addressable”
  - Most modern ISAs are byte addressable, too
  - byte, half words, words are aligned

<table>
<thead>
<tr>
<th>Byte addresses</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td></td>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td></td>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td></td>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>0xFFFE</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0xFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Half Word Addr</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td></td>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x0006</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0xFFFc</td>
<td>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Word Addresses</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>0xFFFc</td>
<td>.</td>
</tr>
</tbody>
</table>
R-type

- op \$rd, \$rs, \$rt
  - 3 regs.: add, addu, and, nor, or, sltu, sub, subu
  - 2 regs.: sll, srl
  - 1 reg.: jr

- 1 arithmetic operation, 1 I-memory access

Example:
  
  \[
  \begin{array}{c}
  \text{opcode} \ 0x0 \ 
  \text{rs} \ 0x0 \ 
  \text{rt} \ 0x0 \ 
  \text{rd} \ 0x0 \ 
  \text{shamt} \ 0x0 \ 
  \text{funct} \ 0x20
  \end{array}
  \]

- `sll \$t0, \$t1, 8: R[8] = R[9] << 8`
  
  \[
  \begin{array}{c}
  \text{opcode} \ 0x0 \ 
  \text{rs} \ 0x0 \ 
  \text{rt} \ 0x0 \ 
  \text{rd} \ 0x0 \ 
  \text{shamt} \ 0x0 \ 
  \text{funct} \ 0x0
  \end{array}
  \]
I-type

- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access
- Example:
  - lw $s0, 4($s2):
  - lw $s0, 0($s2)
  - add $s2, $s2, $s1
I-type (cont.)

- op $rt, $rs, immediate
  - addi, addiu, andi, beq, bne, ori, slti, sltiu
- op $rt, offset($rs)
  - lw, lbu, lhu, ll, lui, sw, sb, sc, sh
- 1 arithmetic op, 1 I-memory and 1 D-memory access

**Example:**
- beq $t0, $t1, -40
  
  \[
  \text{if } (R[8] == R[9]) \text{ PC = PC + 4 + 4*(-40)}
  \]
J-type

- **op immediate**
  - j, jal
- **1 instruction memory access, 1 arithmetic op**
- **Example:**
  - jal quicksort:
    - R[31] = PC + 4
    - PC = quicksort
Practice

- Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

Assume int is 32 bits

$s0 = &A[0]$

$\text{v0} = \text{sum}$

$t0 = i$

```
and $t0, t0, \text{zero} \# \text{let i = 0}
addi t1, \text{zero}, 100 \# \text{temp = 100}
lw t3, 0($s0) \# \text{temp1 = A[i]}
add v0, v0, t3 \# \text{sum += temp1}
addi s0, s0, 4 \# \text{addr of A[i+1]}
addi t0, t0, 1 \# i = i+1
bne t1, t0, LOOP \# if i < 100

1. Initialization
2. Load A[i] from memory to register
3. Add the value of A[i] to sum
4. Increase by 1
5. Check if i still < 100```
Tower of Hanoi

int hanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 2*hanoi(n-1)+1;
}

int main(int argc, char **argv)
{
    int n, result;
    n = atoi(argv[0]);
    result = hanoi(n);
    printf("%d\n", result);
}
Function calls

- Passing arguments
  - $a0-$a3
  - more to go using the memory stack
- Invoking the function
  - jal <label>
  - store the PC of jal +4 in $ra
- Return value in $v0
- Return to caller
  - jr $ra
Let’s write the hanoi()

```c
int hanoi(int n)
{
    if(n==1)
        return 1;
    else
        return 2*hanoi(n-1)+1;
}
```

hanoi: addi $a0, $a0, -1  // n = n-1
       bne $a0, $zero, hanoi_1  // if(n == 0) goto: hanoi_1
       addi $v0, $zero, 1  // return_value = 0 + 1 = 1
       j return  // return
hanoi_1: jal hanoi  // call hanoi
       sll $v0, $v0, 1  // return_value=return_value*2
       addi $v0, $v0, 1  // return_value = return_value+1
return: jr $ra  // return to caller
Function calls

Caller (main)  

Prepare argument for hanoi
$a0 - a3$ for passing arguments

Callee (hanoi)

addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
add $t0, $zero, $a0
li $v0, 4
syscall

hanoi: addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

hanoi_1: jal hanoi

sll $v0, $v0, 1
addi $v0, $v0, 1
return: jr $ra

hanoi: addi $a0, $a0, -1
bne $a0, $zero, hanoi_1
addi $v0, $zero, 1
j return

hanoi_1: jal hanoi

sll $v0, $v0, 1
addi $v0, $v0, 1
return: jr $ra

PC1: jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
add $t0, $zero, $a0
li $v0, 4
syscall

Where are we going now?

Overwrite!

Point to PC1+4

ra | hanoi_1+4 |
Manage registers

- **Sharing registers**
  - A called function will modified registers
  - The caller may use these values later

- **Using memory stack**
  - The stack provides local storage for function calls
  - FILO (first-in-last-out)
  - For historical reasons, the stack grows from high memory address to low memory address
  - The stack pointer ($sp) should point to the top of the stack
Function calls

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $a0, $t1, $t0</td>
<td>hanoi: addi $sp, $sp, -8</td>
</tr>
<tr>
<td>addi $v0, $v0, 1</td>
<td>hanoi_0: addi $a0, $a0, -1</td>
</tr>
<tr>
<td>add $t0, $zero, $a0</td>
<td>bne $a0, $zero, hanoi_1</td>
</tr>
<tr>
<td>li $v0, 4</td>
<td>addi $v0, $zero, 1</td>
</tr>
<tr>
<td>syscall</td>
<td>j return</td>
</tr>
</tbody>
</table>

hanoi:

- save shared registers to the stack, maintain the stack pointer
- restore shared registers from the stack, maintain the stack pointer
Recursive calls

**Caller**

```assembly
... 
addi $a0, $zero, 2
addi $a0, $t1, $t0
jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
li $v0, 4
syscall
... 
```

**Callee**

```assembly
hanoi: addi $sp, $sp, -8
       sw $ra, 0($sp)
       sw $a0, 4($sp)
hanoi_0:addi $a0, $a0, -1
       bne $a0, $zero, hanoi_1
       addi $v0, $zero, 1
       j return
hanoi_1:jal hanoi
sll $v0, $v0, 1
addi $v0, $v0, 1
return: lw $a0, 4(sp)
       lw $ra, 0(sp)
       addi $sp, $sp, 8
       jr $ra
... 
```
Uniformity of MIPS

- Only 3 instruction formats
  - opcodes, rs, rt, immediate are always at the same place
- Similar amounts of work per instruction
  - only 1 read from instruction memory
  - \( \leq 1 \) arithmetic operations
  - \( \leq 2 \) register reads, \( \leq 1 \) register write
  - \( \leq 1 \) data memory access
- Fixed instruction length
- Relatively large register file: 32 registers
- Reasonably large immediate field: 16 bits
- Wise use of opcode space: only 6 bit, R-type get another 6
x86
x86

- The most widely used ISA
- A poorly-designed ISA
  - It breaks almost every rule of a good ISA
    - variable length of instructions
    - the work of each instruction is not equal
    - makes the hardware become very complex
  - It’s popular ≠ It’s good
- You don’t have to know how to write it, but you need to be able to read them and compare x86 with other ISAs
- Reference
# x86 Registers

<table>
<thead>
<tr>
<th>16bit</th>
<th>32bit</th>
<th>64bit</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>EAX</td>
<td>RAX</td>
<td>The accumulator register</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td>EBX</td>
<td>RBX</td>
<td>The base register</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td>ECX</td>
<td>RCX</td>
<td>The counter</td>
<td></td>
</tr>
<tr>
<td>DX</td>
<td>EDX</td>
<td>RDX</td>
<td>The data register</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
<td>Stack pointer</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
<td>Pointer to the base of stack frame</td>
<td></td>
</tr>
<tr>
<td>Rn</td>
<td>RnD</td>
<td>RnD</td>
<td>General purpose registers (8-15)</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
<td>Source index for string operations</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
<td>Destination index for string operations</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>EIP</td>
<td>RIP</td>
<td>Instruction pointer</td>
<td></td>
</tr>
<tr>
<td>FLAGS</td>
<td></td>
<td></td>
<td>Condition codes</td>
<td></td>
</tr>
</tbody>
</table>

These can be used more or less interchangeably.
MOV and addressing modes

- MOV instruction can perform load/store as in MIPS
- MOV instruction has many address modes
  - an example of non-uniformity

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $6, %eax</td>
<td>R[eax] = 0x6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl .L0, %eax</td>
<td>R[eax] = .L0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl %ebx, %eax</td>
<td>R[ebx] = R[eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>movl -4(%.ebp), %ebx</td>
<td>R[ebx] = mem[R[ebp]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl (%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>movl -4(%ecx,%eax,4), %eax</td>
<td>R[eax] = mem[R[ebx]+R[edx]*4-4]</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>movl %ebx, -4(%.ebp)</td>
<td>mem[R[ebp]-4] = R[ebx]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>movl $6, -4(%.ebp)</td>
<td>mem[R[ebp]-4] = 0x6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Arithmetic Instructions

- Accepts memory addresses as operands
- Register-memory ISA

<table>
<thead>
<tr>
<th>instruction</th>
<th>meaning</th>
<th>arithmetic op</th>
<th>memory op</th>
</tr>
</thead>
<tbody>
<tr>
<td>subl $16, %esp</td>
<td>R[%esp] = R[%esp] - 16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl %eax, %esp</td>
<td>R[%esp] = R[%esp] - R[%eax]</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>subl -4(%ebx), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]-4]</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>subl (%ebx, %edx, 4), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]+R[edx]*4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl -4(%ebx, %edx, 4), %eax</td>
<td>R[eax] = R[eax] - mem[R[ebx]+R[edx]*4-4]</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>subl %eax, -4(%ebx)</td>
<td>mem[R[ebx]-4] = mem[R[ebx]-4]-R[eax]</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Branch instructions

• x86 use condition codes for branches
  • Arithmetic instruction sets the flags
  • Example:
    
    ```
    cmp %eax, %ebx #computes %eax-%ebx, sets the flag
    je <location>  #jump to location if equal flag is set
    ```

• Unconditional branches
  • Example:
    
    ```
    jmp <location>  #jump to location
    ```
Summation for x86

• Translate the C code into assembly:

```c
for(i = 0; i < 100; i++)
{
    sum+=A[i];
}
```

```assembly
xorl %eax, %eax
.L2: addl (%ecx,%eax,4), %edx
    addl $1, %eax
    cmpl $100, %eax
    jne .L2
Assume
int is 32 bytes
%ecx = &A[0]
%edx = sum;
%eax = i;
```
## MIPS v.s. x86

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA type</td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td>instruction width</td>
<td>32 bits</td>
<td>1 ~ 17 bytes</td>
</tr>
<tr>
<td>code size</td>
<td>larger</td>
<td>smaller</td>
</tr>
<tr>
<td>registers</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>addressing modes</td>
<td>reg+offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+offset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>base+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scaled+index+offset</td>
</tr>
<tr>
<td>hardware</td>
<td>simple</td>
<td>complex</td>
</tr>
</tbody>
</table>
Translate from C to Assembly

- `gcc`: `gcc [options] [src_file]`
  - compile to binary
    - `gcc -o foo foo.c`
  - compile to assembly (assembly in `foo.s`)
    - `gcc -S foo.c`
  - compile with debugging message
    - `gcc -g -S foo.c`
  - optimization
    - `gcc -On -S foo.c`
      - n from 0 to 3 (0 is no optimization)
gdb: GNU DeBugger

- **gdb**: gdb executable_filename
  - the executable file must be compiled with -g
  - (gdb) run [arguments]
    - start running the program
- create breakpoints:
  - (gdb) break source_filename:line_number
  - (gdb) break source_filename:function_name()
  - (gdb) break *PC
gdb: GNU DeBugger

- display breakpoints
  - (gdb) info breakpoints
- enable/disable breakpoints
  - (gdb) enable/disable breakpoint_number
- remove breakpoints
  - (gdb) delete breakpoint_number
  - (gdb) clear source_filename:line_number
gdb: GNU DeBugger

- Inspect values:
  - `print variable_name/register_name`
  - `info registers`
gdb: GNU DeBugger

- Step through program
  - s: step to the next line in source code
  - si: step to the next machine instruction
  - n: step over function.
Other than MIPS & x86
ISA alternative

- MIPS is a 3-address ISA
- 2-address ISA
  - add $t1, $t2: \( R[\$t1] = R[\$t1] + R[\$t2] \)
  - pros: fewer operands, shorter instructions
  - cons: lots of extra memory copies
- 1-address ISA: accumulator
  - add $t1: accu = accu + R[\$t1]  
- 0-address ISA: stack-based ISA
  - add: \( t1 = \text{pop} \), \( t2 = \text{pop} \), \( t3 = t1 + t2 \), push
## Different types of ISA

<table>
<thead>
<tr>
<th></th>
<th>stack</th>
<th>accumulator</th>
<th>register-memory</th>
<th>load-store</th>
</tr>
</thead>
<tbody>
<tr>
<td>addresses</td>
<td>0</td>
<td>1</td>
<td>2 or 3</td>
<td>3</td>
</tr>
<tr>
<td>A=X<em>Y-B</em>C</td>
<td>push B</td>
<td>load B</td>
<td>R1 = X*Y</td>
<td>load t1, X</td>
</tr>
<tr>
<td></td>
<td>push C</td>
<td>mul C</td>
<td>R2 = B*C</td>
<td>load t2, Y</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>store temp</td>
<td>A = R1-R2</td>
<td>mul t2, t1, t2</td>
</tr>
<tr>
<td></td>
<td>push X</td>
<td>load X</td>
<td></td>
<td>load t3, B</td>
</tr>
<tr>
<td></td>
<td>push Y</td>
<td>mul Y</td>
<td></td>
<td>load t4, C</td>
</tr>
<tr>
<td></td>
<td>mul</td>
<td>sub temp</td>
<td></td>
<td>mul t4, t4, t3</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>store A</td>
<td></td>
<td>sub t4, t3, t4</td>
</tr>
<tr>
<td></td>
<td>pop A</td>
<td></td>
<td></td>
<td>store t4, A</td>
</tr>
</tbody>
</table>

### Additional Notes:

**+**
- high code density
- easy to compile
- short instructions
- fewest instructions
- simple hardware
- fewest memory access

**-**
- hardware stack design
- most memory access
- complex hardware design
- code size
Stack-based ISA

- A push-down stack holds arguments
- Some instructions manipulate the stack
- Most instructions works on stack
  - zero-operand instructions
- Elegant in theory
- Clumsy in hardware
  - how to design the stack?
- Example:
  - Java Virtual machine
  - x86 floating point