Instruction Set Architecture

Hung-Wei Tseng
Recap: computer architecture

• The art or science of building computers
  • Processors and memories are everywhere in computers
  • We care about performance, power, cost, and etc. when building computers

• Moore’s law still allows us to shrink the die size, but power consumption limits the single chip/core performance

• We are now in the era of multicore processors, but parallelizing a program to utilize multithreaded processors is not easy
Outline

• How we talk to computers
• How to design an ISA (instruction set architecture)
• MIPS ISA
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
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
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 operand?
  - What type of operands?
    - Memory/register/label/number (immediate value)

- Format
  - Length
  - Formats?
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
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>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>0x0</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>0x0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0x0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x0</td>
</tr>
<tr>
<td>0x0008</td>
<td>0x0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0x0</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:**
    - opcode = 0x0, funct = 0x20
  - `sll $t0, $t1, 8: R[8] = R[9] << 8`
    - opcode = 0x0, shamt = 0x8, funct = 0x0

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

23
### I-type

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

- **op $rt, $rs, immediate**
  - addi, addiu, andi, beq, bne, ori, s1ti, s1tiu
- **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):
    \[ R[16] = \text{mem}[R[18]+4] \]
  - add $s2, $s2, $s1
  - lw $s0, 0($s2)

Only two addressing modes.
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 memory access
- Example:
  - beq $t0, $t1, -40
    if (R[8] == R[9]) PC = PC + 4 + 4*(-40)
J-type

- op immediate
  - j, jal
- 1 instruction memory access, 1 arithmetic op
- Example:
  - jal quicksort:
    - $R[31] = PC + 4$
    - $PC = \text{quicksort}$
• Translate the C code into assembly:

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

Assume int is 32 bits
$s0 = &A[0]$

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
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()

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 honai
           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
```

**After hanoi**

```
ra | hanoi_1+4 |
```

**PC1:**
- jal hanoi

```
sll $v0, $v0, 1
addi $v0, $v0, 1
j    return
hanoi_1:jal hanoi
```

**Overwrite!**

```
return: jr $ra
```

**Where are we going now?**

```
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
```

**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

**Caller**

- \textbf{addi} \textit{a0}, \textit{t1}, \textit{t0}

**Callee**

- \textbf{hanoi}: \textit{addi} \textit{sp}, \textit{sp}, -8  
  \textbf{sw} \textit{ra}, 0(\textit{sp})  
  \textbf{sw} \textit{a0}, 4(\textit{sp})

- \textbf{hanoi}_0: \textit{addi} \textit{a0}, \textit{a0}, -1  
  \textbf{bne} \textit{a0}, \textit{zero}, \textit{hanoi}_1  
  \textbf{addi} \textit{v0}, \textit{zero}, 1  
  \textbf{j} \textit{return}

- \textbf{hanoi}_1: \textbf{jal} \textit{hanoi}

- \textbf{add} \textit{t0}, \textit{zero}, \textit{a0}

- \textbf{li} \textit{v0}, 4

- \textbf{syscall}

- \textbf{return}: \textbf{jw} \textit{ra}, 4(\textit{sp})  
  \textbf{lw} \textit{ra}, 0(\textit{sp})  
  \textbf{addi} \textit{sp}, \textit{sp}, 8  
  \textbf{jr} \textit{ra}

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
```
Demo

• The overhead of function calls
• The keyword `inline` in C can embed the callee code at the call site
  • Eliminates function call overhead
• Does not work if it’s called using a function pointer
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
  • http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
## 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>
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>ISA type</th>
<th>MIPS</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<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>base+offset, base+index, scaled+index, 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)
Demo

- The magic of compiler optimization!
- Without optimization
- After compiled with -O3
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 = pop, t2 = 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>store t4, A</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>•high code density •easy to compile</td>
<td>•short instructions</td>
<td>•fewest instructions</td>
<td>•simple hardware •fewest memory access</td>
</tr>
<tr>
<td>-</td>
<td>•hardware stack design</td>
<td>•most memory access</td>
<td>•complex hardware design</td>
<td>•code size</td>
</tr>
</tbody>
</table>
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