Computer Architecture: A Short Introduction

Alexandre Bartel

2019

What is a computer?

- Input devices (keyboard, mouse, ...)
- Processing units (CPU, ...)
- Output devices (screen, ...)

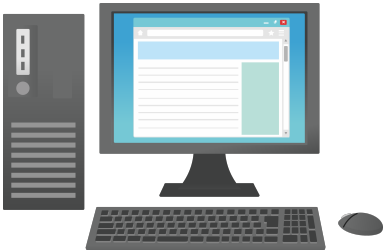


image source: openclipart.com

Inside a Computer

1. Screen
2. Motherboard
3. Central Processing Unit (CPU)
4. Memory
5. Extension cards
6. Power Supply
7. Bluray reader
8. Hard drive
9. Keyboard
10. Mouse

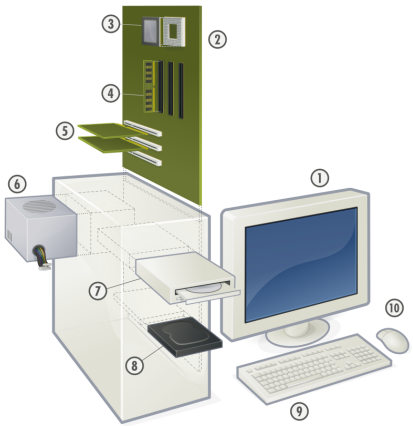


image source: wikimedia.org

Components Identification

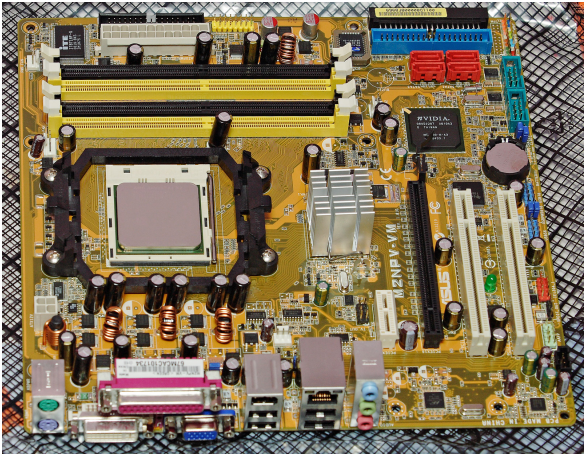


image source: wikimedia.org

Components Identification

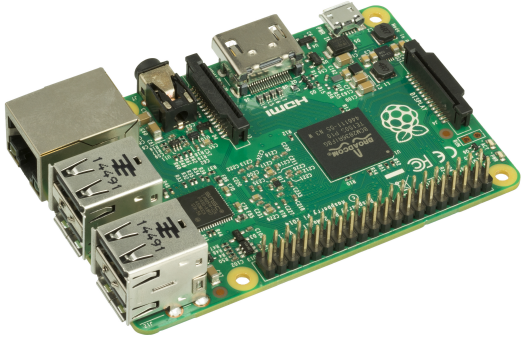
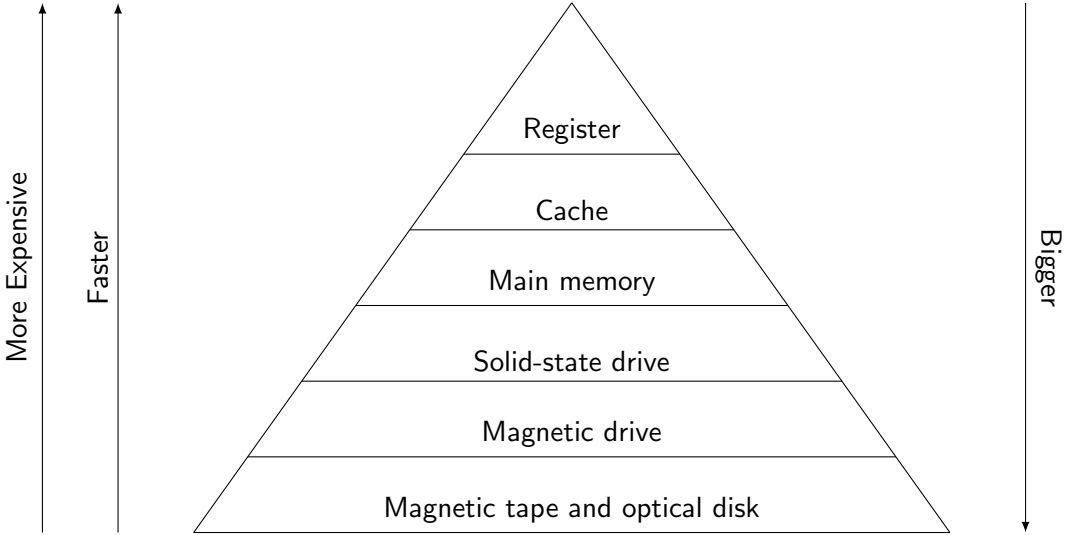


image source: wikimedia.org

Why Multiple Storage Devices?



Layers from Applications to the Hardware

Applications	(Web Browser, Compiler, Text editor, ...)
Operating System	(GNU/Linux, NetBSD, Windows, MacOS, ...)
Hardware	(CPU, Memory, I/O, ...)

The layer between the software and the hardware is called the Instruction Set Architecture (ISA).

ISA (Instruction Set Architecture)

- Is an abtract model of a computer
- A.k.a. computer architecture
- Defines the set of available machine instructions (ex: MOV, POP)
- Defines the location of data (ex: registers, main memory)
- Two main categories
  - Complex Instruction Set Computer (CISC) such as Motorola 6800, Intel 8080, Intel x86 32-bit, Inter x86 64-bit, ...
  - Reduced Instruction Set Computer (RISC) such as ARM, Atmel AVR, Blackfin, i860, i960, M88000, MIPS, PA-RISC, ...
- Compatibility for a given ISA enables to reuse binaries

ISA in SVEM

In SVEM we use Intel's x86 64-bit ISA.

Where is the ARM's ISA used quite a lot nowadays?

x86 64-bit CPU Registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

x86 64-bit CPU Registers (cont)

- rip (instruction pointer, 32-bit called eip).
- rflags register (32-bit called flags)
- floating-point registers: eight 80-bit x87 registers: ST(0) to ST(7), eight 64-bit MMX registers: MMX0 to MMX7. (these last 8 registers overlap with the x87 registers.)
- sixteen 128-bit SSE [1] registers (8 for 32-bit): XMM0 to XMM15

[1] Streaming SIMD Extensions (SSE) is a single instruction, multiple data (SIMD) instruction set extension. A SIMD performs the same operation on multiple data points at the same time, thus it potentially reduces the execution time of the program.

Adressing Modes

-

Compiling & Executing a Program

- Generaly a program is writen in a "high-level" language (ex: C, C++, Java)
- The program is compiled to machine code
- The program is executed by the CPU
- Any operation (ex: addition) is performed using CPU registers

Source Code, Machine Code & Assembly Code

```
int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
  }
  return sum;
}
```

- Source code (top left) and assembly code (right) are understandable for a human
- Machine code (middle) is understandable for a processor

```
0000000000001125 <main>:
    1125:   55                      push   %rbp
    1126:   48 89 e5                mov    %rsp,%rbp
    1129:   48 83 ec 08             sub    $0x8,%rsp
    112d:   89 7d 8c                mov    %edi,-0x74(%rbp)
    1130:   48 89 75 80             mov    %rsi,-0x80(%rbp)
    1134:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
    113b:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
    1142:   eb 14                   jmp    1158 <main+0x33>
    1144:   8b 45 f8                mov    -0x8(%rbp),%eax
    1147:   48 98                   cltq
    1149:   0f b6 44 05 90          movzbl -0x70(%rbp,%rax,1),%eax
    114e:   0f be c0                movsbl %al,%eax
    1151:   01 45 fc                add    %eax,-0x4(%rbp)
    1154:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
    1158:   83 7d f8 63             cmpl   $0x63,-0x8(%rbp)
    115c:   7e e6                   jle    1144 <main+0x1f>
    115e:   8b 45 fc                mov    -0x4(%rbp),%eax
    1161:   c9                      leaveq
    1162:   c3                      retq
    1163:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    116a:   00 00 00
    116d:   0f 1f 00                nopl   (%rax)
```

A Note on Assembly Syntax (Intel vs. AT&T

	Intel	AT&T
Sigils	assembler detects type of symbols	Immediate values prefixed with a "$", registers prefixed with "%"
Parameter Order	ex: mov rsp, 42	ex: mov $42, %rsp
Parameter Size	devrived from the register name: 64-, 32-, 16-, 8-bit imply q, l, w, b, respectively ex: add ebx, 3	mnemonics are suffixed by a letter: 'q' for qword, 'l' for long (dword), 'w' for word, and 'b' for byte ex: addl $3, %ebx
Effective Address	mov rax, [ecx + ebx*31 + memory_location] size keywords like byte, word, or dword have to be used if the size cannot be determined from the operands ex: movzb eax, [rbp + rax*1 - 0x70]	movq memory_location(%ecx,%ebx,31), %rax DISPLACEMENT(BASE,INDEX,SCALE) ex: movzbl -0x70(%rbp,%rax,1),%eax

Stack, RSP

```c
int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
  }
  return sum;
}
```

- function parameters: `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`. This is the calling convention of the System V AMD64 Application Binary Interface (ABI).
- Variables `sum`, `i` and `elements` are stored on the stack ?
- We do not see that 100 bytes have been reserved
- Return value: `eax`

```
0000000000001125 <main>:
    1125:   push   %rbp
    1126:   mov    %rsp,%rbp
    1129:   sub    $0x8,%rsp
    112d:   mov    %edi,-0x74(%rbp)
    1130:   mov    %rsi,-0x80(%rbp)
    1134:   movl   $0x0,-0x4(%rbp)
    113b:   movl   $0x0,-0x8(%rbp)
    1142:   jmp    1158 <main+0x33>
    1144:   mov    -0x8(%rbp),%eax
    1147:   cltq
    1149:   movzbl -0x70(%rbp,%rax,1),%eax
    114e:   movsbl %al,%eax
    1151:   add    %eax,-0x4(%rbp)
    1154:   addl   $0x1,-0x8(%rbp)
    1158:   cmpl   $0x63,-0x8(%rbp)
    115c:   jle    1144 <main+0x1f>
    115e:   mov    -0x4(%rbp),%eax
    1161:   leaveq
    1162:   retq
    1163:   nopw   %cs:0x0(%rax,%rax,1)
    116a:
    116d:   nopl   (%rax)
```

Stack, RSP

```c
int function() { }

int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
    function();
  }
  return sum;
}
```

- main now calls function
- now, we can see that 100+ bytes have been "reserved"

```
000000000000112c <main>:
    112c        push   %rbp
    112d        mov    %rsp,%rbp
    1130        add    $0xffffffffffffff80,%rsp
    1134        mov    %edi,-0x74(%rbp)
    1137        mov    %rsi,-0x80(%rbp)
    113b        movl   $0x0,-0x4(%rbp)
    1142        movl   $0x0,-0x8(%rbp)
    1149        jmp    1169 <main+0x3d>
    114b        mov    -0x8(%rbp),%eax
    114e        cltq
    1150        movzbl -0x70(%rbp,%rax,1),%eax
    1155        movsbl %al,%eax
    1158        add    %eax,-0x4(%rbp)
    115b        mov    $0x0,%eax
    1160        callq  1125 <function>
    1165        addl   $0x1,-0x8(%rbp)
    1169        cmpl   $0x63,-0x8(%rbp)
    116d        jle    114b <main+0x1f>
    116f        mov    -0x4(%rbp),%eax
    1172        leaveq
    1173        retq
    1174        nopw   %cs:0x0(%rax,%rax,1)
```

Optimization 1

```c
int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
  }
  return sum;
}
```

- Uses registers instead of the stack
- gcc -O1

```
0000000000001126 <main>:
    1126:       lea    -0x70(%rsp),%rdx
    112b:       lea    0x64(%rdx),%rsi
    112f:       mov    $0x0,%eax
    1134:       movsbl (%rdx),%ecx
    1137:       add    %ecx,%eax
    1139:       add    $0x1,%rdx
    113d:       cmp    %rsi,%rdx
    1140:       jne    1134 <main+0xe>
    1142:       retq
    1143:       nopw   %cs:0x0(%rax,%rax,1)
    114a:
    114d:       nopl   (%rax)
```

Optimization 2

```c
int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
  }
  return sum;
}
```

- Improve speed by changing some instructions
- gcc -O2

```
0000000000001040 <main>:
    1040        lea    -0x78(%rsp),%rdx
    1045        xor    %eax,%eax
    1047        lea    0x64(%rdx),%rsi
    104b        nopl   0x0(%rax,%rax,1)
    1050        movsbl (%rdx),%ecx
    1053        add    $0x1,%rdx
    1057        add    %ecx,%eax
    1059        cmp    %rdx,%rsi
    105c        jne    1050 <main+0x10>
    105e        retq
    105f        nop
```

Optimization 1 vs. Optimization 2

```
0000000000001126 <main>:
    1126        lea    -0x70(%rsp),%rdx
    112b        lea    0x64(%rdx),%rsi
    112f        mov    $0x0,%eax
    1134        movsbl (%rdx),%ecx
    1137        add    %ecx,%eax
    1139        add    $0x1,%rdx
    113d        cmp    %rsi,%rdx
    1140        jne    1134 <main+0xe>
    1142        retq
    1143        nopw   %cs:0x0(%rax,%rax,1)
    114a
    114d        nopl   (%rax)
```

```
0000000000001040 <main>:
    1040        lea    -0x78(%rsp),%rdx
    1045        xor    %eax,%eax
    1047        lea    0x64(%rdx),%rsi
    104b        nopl   0x0(%rax,%rax,1)
    1050        movsbl (%rdx),%ecx
    1053        add    $0x1,%rdx
    1057        add    %ecx,%eax
    1059        cmp    %rdx,%rsi
    105c        jne    1050 <main+0x10>
    105e        retq
    105f        nop
```

Optimization 3

```
int main(int argc, char** argv) {
  char elements[100]; int sum = 0;
  for (int i = 0; i < 100; i++) {
    sum += elements[i];
  }
  return sum;
}
```

- Speed optimization
- gcc -O3
- (full assembly code on the next slide)
- What is the "idea" behind this optimization?

```
0000000000001040 <main>:
    1040:   pxor   %xmm2,%xmm2
    1044:   movdqa -0x78(%rsp),%xmm3
    104a:   movsbl -0x18(%rsp),%edx
    104f:   movdqa %xmm2,%xmm1
    1053:   pcmpgtb %xmm3,%xmm1
    1057:   movdqa %xmm3,%xmm0
    105b:   punpcklbw %xmm1,%xmm0
    105f:   punpckhbw %xmm1,%xmm3
    1063:   pxor   %xmm1,%xmm1
    1067:   movdqa %xmm1,%xmm4
    106b:   movdqa %xmm0,%xmm5
    106f:   pcmpgtw %xmm0,%xmm4
    1073:   punpcklwd %xmm4,%xmm5
    1077:   punpckhwd %xmm4,%xmm0
    107b:   movdqa %xmm1,%xmm4
    107f:   pcmpgtw %xmm3,%xmm4
    1083:   paddd  %xmm5,%xmm0
    1087:   movdqa %xmm3,%xmm5
    108b:   punpcklwd %xmm4,%xmm5
    108f:   punpckhwd %xmm4,%xmm3
    [...]
```

Optimization 3 (cont)

```
0000000000001040 <main>:
    1040:	pxor   %xmm2,%xmm2
    1044:	movdqa -0x78(%rsp),%xmm3
    104a:	movsbl -0x18(%rsp),%edx
    104f:	movdqa %xmm2,%xmm1
    1053:	pcmpgtb %xmm3,%xmm1
    1057:	movdqa %xmm3,%xmm0
    105b:	punpcklbw %xmm1,%xmm0
    105f:	punpckhbw %xmm1,%xmm3
    1063:	pxor   %xmm1,%xmm1
    1067:	movdqa %xmm1,%xmm4
    106b:	movdqa %xmm0,%xmm5
    106f:	pcmpgtw %xmm0,%xmm4
    1073:	punpcklwd %xmm4,%xmm5
    1077:	punpckhwd %xmm4,%xmm0
    107b:	movdqa %xmm1,%xmm4
    107f:	pcmpgtw %xmm3,%xmm4
    1083:	paddd  %xmm5,%xmm0
    1087:	movdqa %xmm3,%xmm5
    108b:	punpcklwd %xmm4,%xmm5
    108f:	punpckhwd %xmm4,%xmm3
    1093:	paddd  %xmm5,%xmm0
    1097:	movdqa %xmm2,%xmm5
    109b:	paddd  %xmm3,%xmm0
    109f:	movdqa -0x68(%rsp),%xmm3
    10a5:	pcmpgtb %xmm3,%xmm5
    10a9:	movdqa %xmm3,%xmm4
    10ad:	punpcklbw %xmm5,%xmm4
    10b1:	punpckhbw %xmm5,%xmm3
    10b5:	movdqa %xmm1,%xmm5
    10b9:	pcmpgtw %xmm4,%xmm5
    10bd:	movdqa %xmm4,%xmm6
    10c1:	punpcklwd %xmm5,%xmm6
    10c5:	punpckhwd %xmm5,%xmm4
    10c9:	movdqa %xmm3,%xmm5
    10cd:	paddd  %xmm6,%xmm0
    10d1:	paddd  %xmm4,%xmm0
    10d5:	movdqa %xmm1,%xmm4
    10d9:	pcmpgtw %xmm3,%xmm4
    10dd:	punpcklwd %xmm4,%xmm5
    10e1:	punpckhwd %xmm4,%xmm3
    10e5:	paddd  %xmm5,%xmm0
    10e9:	movdqa %xmm2,%xmm5
    10ed:	paddd  %xmm3,%xmm0
    10f1:	movdqa -0x58(%rsp),%xmm3
    10f7:	pcmpgtb %xmm3,%xmm5
    10fb:	movdqa %xmm3,%xmm4
    10ff:	punpcklbw %xmm5,%xmm4
    1103:	punpckhbw %xmm5,%xmm3
    1107:	movdqa %xmm1,%xmm5
    110b:	pcmpgtw %xmm4,%xmm5
    110f:	movdqa %xmm4,%xmm6
    1113:	punpcklwd %xmm5,%xmm6
    1117:	punpckhwd %xmm5,%xmm4
    111b:	movdqa %xmm3,%xmm5
    111f:	paddd  %xmm6,%xmm0
    1123:	paddd  %xmm4,%xmm0
    1127:	movdqa %xmm1,%xmm4
    112b:	pcmpgtw %xmm3,%xmm4
    112f:	punpcklwd %xmm4,%xmm5
    1133:	punpckhwd %xmm4,%xmm3
    1137:	paddd  %xmm5,%xmm0
    113b:	movdqa %xmm2,%xmm5
    113f:	paddd  %xmm3,%xmm0
    1143:	movdqa -0x48(%rsp),%xmm3
    1149:	pcmpgtb %xmm3,%xmm5
    114d:	movdqa %xmm3,%xmm4
    1151:	punpcklbw %xmm5,%xmm4
    1155:	punpckhbw %xmm5,%xmm3
    1159:	movdqa %xmm1,%xmm5
    115d:	pcmpgtw %xmm4,%xmm5
    1161:	movdqa %xmm4,%xmm6
    1165:	punpcklwd %xmm5,%xmm6
    1169:	punpckhwd %xmm5,%xmm4
    116d:	movdqa %xmm3,%xmm5
    1171:	paddd  %xmm6,%xmm0
    1175:	paddd  %xmm4,%xmm0
    1179:	movdqa %xmm1,%xmm4
    117d:	pcmpgtw %xmm3,%xmm4
    1181:	punpcklwd %xmm4,%xmm5
    1185:	punpckhwd %xmm4,%xmm3
    1189:	paddd  %xmm5,%xmm0
    118d:	movdqa %xmm2,%xmm5
    1191:	paddd  %xmm3,%xmm0
    1195:	movdqa -0x38(%rsp),%xmm3
    1199:	pcmpgtb %xmm3,%xmm5
    119f:	movdqa %xmm3,%xmm4
    11a3:	punpcklbw %xmm5,%xmm4
    11a7:	punpckhbw %xmm5,%xmm3
    11ab:	movdqa %xmm1,%xmm5
    11af:	pcmpgtw %xmm4,%xmm5
    11b3:	movdqa %xmm4,%xmm6
    11b7:	punpcklwd %xmm5,%xmm6
    11bb:	punpckhwd %xmm5,%xmm4
    11bf:	movdqa %xmm3,%xmm5
    11c3:	paddd  %xmm6,%xmm0
    11c7:	paddd  %xmm4,%xmm0
    11cb:	movdqa %xmm1,%xmm4
    11cf:	pcmpgtw %xmm3,%xmm4
    11d3:	punpcklwd %xmm4,%xmm5
    11d7:	punpckhwd %xmm4,%xmm3
    11db:	paddd  %xmm5,%xmm0
    11df:	paddd  %xmm3,%xmm0
    11e3:	movdqa -0x28(%rsp),%xmm3
    11e9:	pcmpgtb %xmm3,%xmm2
    11ed:	movdqa %xmm3,%xmm4
    11f1:	punpckhbw %xmm2,%xmm3
    11f5:	punpcklbw %xmm2,%xmm4
    11f9:	movdqa %xmm3,%xmm2
    11fd:	movdqa %xmm1,%xmm3
    1201:	movdqa %xmm4,%xmm5
    1205:	pcmpgtw %xmm4,%xmm3
    1209:	pcmpgtw %xmm2,%xmm1
    120d:	punpcklwd %xmm3,%xmm5
    1211:	punpckhwd %xmm3,%xmm4
    1215:	movdqa %xmm2,%xmm3
    1219:	paddd  %xmm5,%xmm0
    121d:	punpcklwd %xmm1,%xmm3
    1221:	punpckhwd %xmm1,%xmm2
    1225:	paddd  %xmm4,%xmm0
    1229:	paddd  %xmm3,%xmm0
    122d:	paddd  %xmm2,%xmm0
    1231:	movdqa %xmm0,%xmm1
    1235:	psrldq $0x8,%xmm1
    123a:	paddd  %xmm1,%xmm0
    123e:	movdqa %xmm0,%xmm1
    1242:	psrldq $0x4,%xmm1
    1247:	paddd  %xmm1,%xmm0
    124b:	movd   %xmm0,%eax
    124f:	add    %edx,%eax
    1251:	movsbl -0x17(%rsp),%edx
    1256:	add    %eax,%edx
    1258:	movsbl -0x16(%rsp),%eax
    125d:	add    %eax,%edx
    125f:	movsbl -0x15(%rsp),%eax
    1264:	add    %edx,%eax
    1266:	retq
    1267:	nopw   0x0(%rax,%rax,1)
    126e
```

Function Call and the Stack

```
int function3() {
  char buffer[0x42];
  return 3;
}
int function2() {
  char buffer[0x42];
  function3();
  return 3;
}
int function1() {
  int a = 2;
  function2();
}
int main(int argc, char** argv) {

  function1();
}
```

- rsp and rbp registers to keep the context
- rsp, Stack Pointer
- rbp, Base Pointer, base address for accessing local variables on the stack

Function Call and the Stack

```
0000000000001125 <function3>:
    1125    push   %rbp
    1126    mov    %rsp,%rbp
    1129    mov    $0x3,%eax
    112e    pop    %rbp
    112f    retq

0000000000001130 <function2>:
    1130    push   %rbp
    1131    mov    %rsp,%rbp
    1134    sub    $0x50,%rsp
    1138    mov    $0x0,%eax
    113d    callq  1125 <function3>
    1142    mov    $0x3,%eax
    1147    leaveq
    1148    retq

0000000000001149 <function1>:
    1149    push   %rbp
    114a    mov    %rsp,%rbp
    114d    sub    $0x10,%rsp
    1151    movl   $0x2,-0x4(%rbp)
    1158    mov    $0x0,%eax
    115d    callq  1130 <function2>
    1162    nop
    1163    leaveq
    1164    retq

0000000000001165 <main>:
    1165    push   %rbp
    1166    mov    %rsp,%rbp
    1169    sub    $0x10,%rsp
    116d    mov    %edi,-0x4(%rbp)
    1170    mov    %rsi,-0x10(%rbp)
    1174    mov    $0x0,%eax
    1179    callq  1149 <function1>
    117e    mov    $0x0,%eax
    1183    leaveq
    1184    retq
    1185    nopw   %cs:0x0(%rax,%rax,1)
    118c
    118f    nop
```

Note: leave <=> mov %rbp, %rsp; pop %rbp;

Note: enter N, vs. leave

Byte Alignment

- The x86-64 System V ABI requires 16 bytes stack alignment before a call.
- For main, the compiler puts the two arguments (argc and argv) on the stack. The first argument is an int (4 bytes) the second a 64-bit pointer (8 bytes). The required size for these arguments is thus 12 bytes. Because of the 16 byte stack alignment, the compiler allocates 16 bytes (0x10)
- For function1, the reasoning is the same with a single int local variable (4 bytes). The compiler allocates 16 bytes (0x10) on the stack instead of 4 to respect the 16 bytes stack alignment.
- For function2, the compiler allocates 80 bytes on the stack (0x50) instead of 66 bytes (0x42) to respect the 16 bytes stack alignment.
- Note that the return address and rbp pushed on the stack form a 16 byte block (already 16 byte aligned)

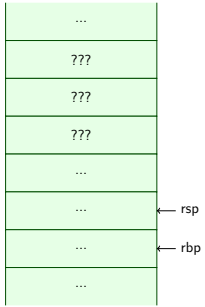
Byte Alignment (cont)

Why Such Alignment?

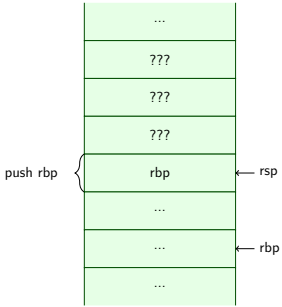
According to the System V AMD64 ABI specifications, the alignment is – at least – required for some instructions:

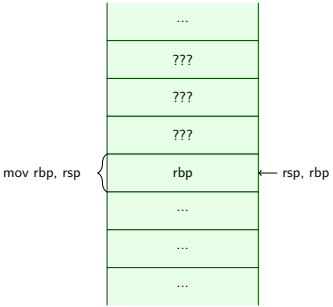
"The alignment requirement allows the use of SSE instructions when operating on the array. The compiler cannot in general calculate the size of a variable-length array (VLA), but it is expected that most VLAs will require at least 16 bytes, so it is logical to mandate that VLAs have at least a 16-byte alignment."

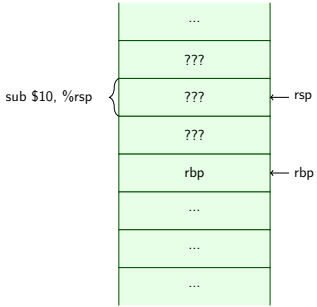
In main...



In main... (cont)


...
???
???
???
push rbp
rbp
⟵ rsp
...
...
⟵ rbp
...


...
???
???
???
mov rbp, rsp
rbp
⟵ rsp, rbp
...
...
...


...
???
sub $10, %rsp
???
⟵ rsp
???
rbp
⟵ rbp
...
...
...

Exercise

Draw the stack at the following points

- Just before function1 is called
- Just before function2 is called
- Just before function3 is called

Von Neumann Architecture vs. Harvard Architecture

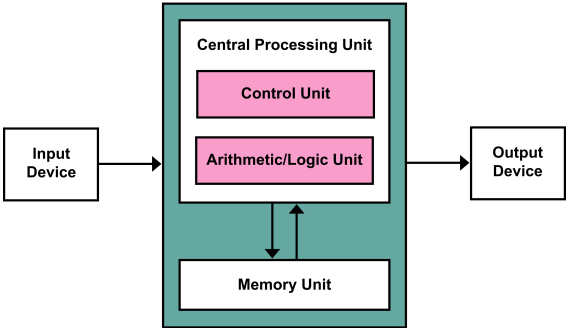


image source: wikimedia.org

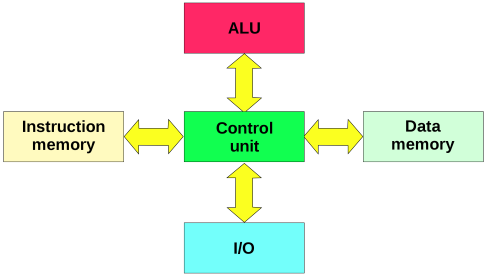


image source: wikimedia.org

Sequential Execution Model

0 Fetch
1 Decode
2 Read Inputs
3 Execute
4 Write Outputs
5 Next Instruction (goto 0)

0 Fetch, 1 Decode

- Length: variable for x86 (max 15 bytes)
- Encoding/Decoding: complex
- Hard to know where next instruction is

2 Read Inputs

- Where does an instruction read data from?
- A register
  - short term memory
  - faster than main memory
  - directly named in instructions
- Main memory
  - longer term memory
  - slower than registers
  - location computer from registers
- Immediate
  - values encoded within instructions

3 Execute

- Manipulates main memory and registers to compute a result

4 Write Outputs

- Where does an instruction read data from?
- A register
  - short term memory
  - faster than main memory
  - directly named in instructions
- Main memory
  - longer term memory
  - slower than registers
  - location computer from registers

5 Next Instruction

- Default: @i + sizeof(i)
- Branching (pc relative, absolute, indirect through register)
- Condition

Question?

Difference of Stack Addresses in Gdb and Outside Gdb

1. Gdb adds environment variables. Since environment variables are stored on the stack, there is a difference in the addresses of local variables inside gdb and outside gdb (because the environment variables gdb adds are not present outside gdb, and thus the stack is shorter outside gdb).
2. Gdb always gives as first argument to the main function, the absolute path to the binary (ex: `/home/user/lab02/test`). If, outside gdb, you start the program with `./test` (because you are in the directory `/home/user/lab02/` containing the binary) you use relative path and thus end up with a shorter string representing the path to the binary. Since this string is stored on the stack it could have an influence of the stack size as well.

Difference of Stack Addresses – Same Environment Variables

What are environement variables?

- Thery are dynamically defined variables that can influence the running of programs.
- For instance, environment variable `PATH`, defines in which directories to search for a binary.

Difference of Stack Addresses – Same Environment Variables (cont)

Print the Value of an Environment Variable

```
$ echo $PATH
/home/user/bin:/usr/local/bin:/usr/bin:/bin
```

Print all Environment Variables

```
$ env
SHELL=/bin/bash
HOME=/home/user
LANG=en_US.UTF-8
TERM=screen
USER=user
PATH=/home/user/bin:/usr/local/bin:/usr/bin:/bin
[...]
```

Difference of Stack Addresses – Same Environment Variables (cont)

Remove all Environment Variables

```
$ env -i
```

Start Program `test` with no Environment Variable

```
$ env -i ./test
```

Start `gdb` with no Environment Variable (gdb adds two env. variables)

```
$ env -i gdb ./test
(gdb) show env
LINES=61
COLUMNS=92
```

Start Program `test` with same Environment as gdb

```
env -i LINES=XX COLUMNS=YY ./test
```

Difference of Stack Addresses – Same Environment Variables (cont)

Print Environment Variables in C

You can add the following code to print environment variables in your C program. That way, you can easily make sure that the environments are the same within and outside gdb.

```c
#include <stdio.h>

extern char** environ;

int main(int argc, char** argv) {
  for (char **env = environ; *env != 0; env++) {
          char *env_var = *env;
          printf("'%s'\n", env_var);
  }
  return 0;
}
```

Difference of Stack Addresses – First Parameter to `main`

First Parameter to `main`

The first parameter to the main function is always the path to the executed binary. You can launch the binary using a relative path (ex: ./test) or an absolute path (ex: /home/user/lab02/test). Gdb always uses the absolute path. To use the absolute path (suppose you are in directory /home/user/lab02) you can execute one of the following commands:

```
$ /home/user/lab02/test
$ `pwd`/test
```

Difference of Stack Addresses – Conclusion

Same Stack Size in gdb and Outside gdb

To launch the program `test` use:

```
$ env -i LINES=XX COLUMNS=YY /home/user/lab02/test
```

To launch gdb use:

```
$ env -i gdb ./test
```

The number of bytes is important, not the content

Note that what is important is the number of bytes of LINES=XX and COLUMNS=YY. If in gdb you have COLUMNS=123, then you should replace COLUMNS=YY by COLUMNS=YYY, otherwise the stacks will not be the same.