https://play.google.com/store/apps/details?id=jp.co.namura.magnifier&hl=zh_HK
https://www.pngfind.com/mpng/iRhTJTh_tux-linux-logo-start-menu-linux-icons-hd/

作業三：瞭解AT&T與gcc的行內組語

中正大學 作業系統實驗室
指導教授：羅習五

圖片來源

- 新垣結衣
  - https://makey.asia/column.php?id=532
  - http://pic.haibao.com/image/14284778.html?kw=%E6%96%B0%E5%9E%A3%E7%BB%93%E8%A1%A3
  - https://huaban.com/pins/835412722/

作業目標及負責助教

- 作業目標：
  - 大概了解x86的組合語言形式
  - 了解gcc所支援的inline assembly的形式
  - 透過這個作業可以讓我們更專心在「組合語言」層級
- 負責助教：
  - 請看網頁

行內組語的形式

asm [volatile]
(AssemblerTemplate //這部分就是組合語言
: OutputOperands // optional，組語會輸出的變數
[: InputOperands // optional，組語會讀取的變數
[: Clobbers] // optional]，組合語言搞爛掉的暫存器的值)

asm.1.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      int a=10;
4.      int b=20;
5.      int c=30;
6.      int d=40;
7.      __asm__ volatile (
8.        "movl $100, %%eax\n"     // eax = 100
9.        "movl $200, %%ebx\n"     // ebx = 200
10.       "addl %%ebx, %%eax\n"    // eax += ebx
11.       "movl %%eax, %0\n"       // b = rax
12.       : "=g" (b)           //output, b的代號是"%0"
13.       : "g" (a), "g" (d)     //input, a代號是"%1", da代號是"%2"
14.       : "ebx", "eax"         //搞髒掉的暫存器，gcc會幫我們還原
15.     );
16.     printf("a=%d, b=%d, c=%d d=%d\n", a, b, c, d);
17. }
```

asm.1.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      int a=10;
4.      int b=20;
5.      int c=30;
6.      int d=40;
7.      __asm__ volatile (
8.        "movl $100, %%eax\n"     // eax = 100
9.        "movl $200, %%ebx\n"     // ebx = 200
10.       "addl %%ebx, %%eax\n"    // eax += rbx
11.       "movl %%eax, %0\n"       // b = ra
12.       : "=g" (b)           //ou
13.       : "g" (a), "g" (d)     //input, a代號是 %1 , dd代號是 %2
14.       : "ebx", "eax"         //搞髒掉的暫存器，gcc會幫我們還原
15.     );
16.     printf("a=%d, b=%d, c=%d d=%d\n", a, b, c, d);
17. }
```

"=" 表示write only，g代表由gcc幫我挑選一個普通暫存器（例如：R0~R31）

asm.1a.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      int a=10;
4.      int b=20;
5.      int c=30;
6.      int d=40;
7.      __asm__ volatile (
8.        "movl $100, %%eax\n"     // eax = 100
9.        "movl $200, %%ebx\n"     // ebx = 200
10.       "addl %%ebx, %%eax\n"    // eax += rbx
11.       "movl %%eax, %0\n"       // b = rax
12.       : "=g" (b)           //output, b的代號是"%0"
13.       : "g" (a), "g" (d)     //input, a代號是
14.       : "ebx", "eax"         //搞髒掉的暫存
15.     );
16.     printf("a=%d, b=%d, c=%d d=%d\n", a, b, c, d);
17. }
```

提示gcc我們改變了ebx和eax的值，如果gcc需要這二個暫存器的舊值，gcc要自己想辦法保存並還原

修飾字

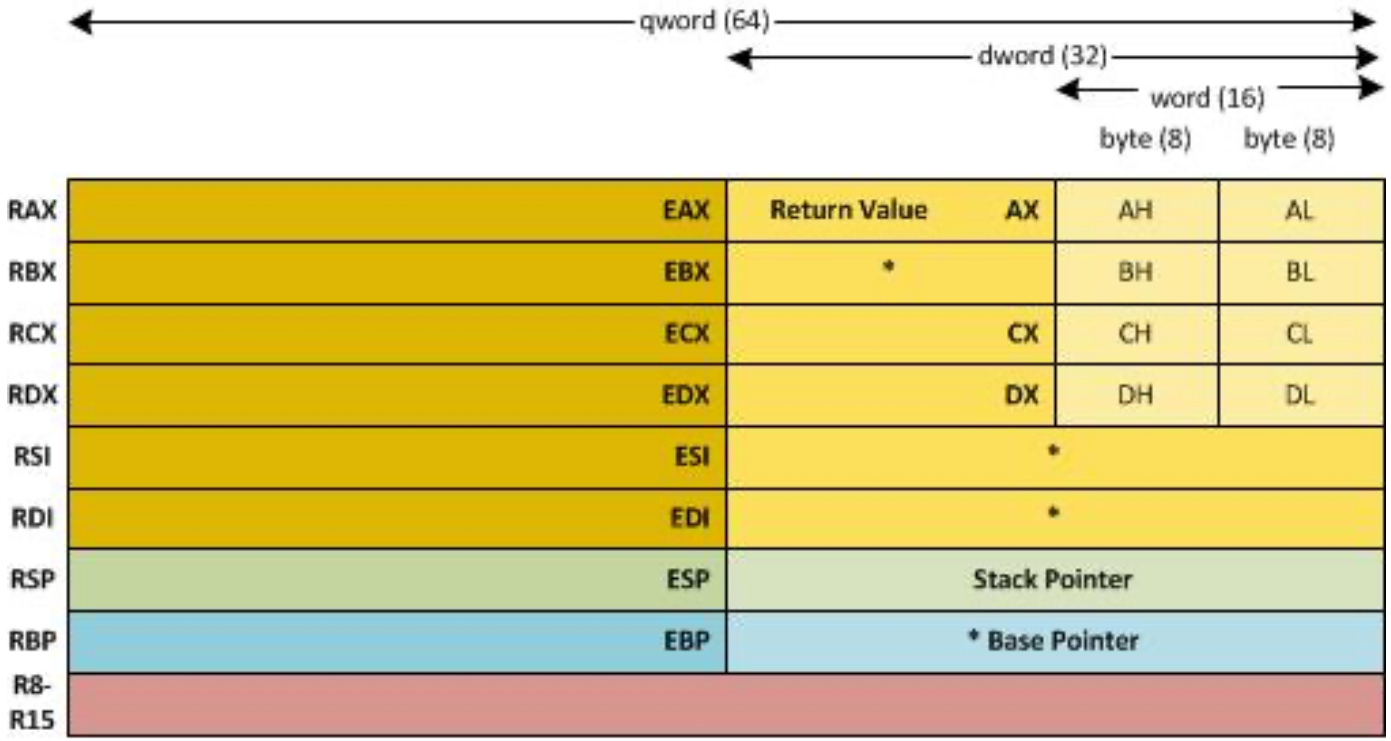
```
  : "=g" (b)
  : "g" (a), "g" (d)
  : "ebx", "eax"
);
```

- Output constraints must begin with either ‘=’ (a variable overwriting an existing value) or ‘+’ (when reading and writing).
- Common constraints include ‘r’ for register and ‘m’ for memory. When you list more than one possible location (for example, "=rm"), the compiler chooses the most efficient one based on the current context.

https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

暫存器長度

- 注意：同一個暫存器會因為名字不同而存取該暫存器的範圍不同
- 例如：實際上RAX、EAX、AX、AH、AL都是同一個暫存器



輸出結果

```
ubuntu@oslab:~/os-lab/sharedFolder$ ./asm.1
a=10, b=300, c=30 d=40
```

asm.1b.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      int a=10;
4.      int b=20;
5.      int c=30;
6.      int d=40;
7.      __asm__ volatile (
8.        "movl %1, %%eax\n"     // eax = a
9.        "movl %2, %%ebx\n"     // ebx = b
10.       "addl %%ebx, %%eax\n"  // eax += rbx
11.       "movl %%eax, %0\n"     // b = rax
12.       : "=g" (b)             //output, b的代號是"%0"
13.       : "g" (a), "g" (d)     //input, a代號是"%1", da代號是"%2"
14.       : "ebx", "eax"         //搞髒掉的暫存器，gcc會幫我們還原
15.     );
16.     printf("a=%d, b=%d, c=%d d=%d\n", a, b, c, d);
17. }
```

自己動手做

- 如果__asm__後面沒有加上**volatile**。那麼這段組合語言可能會被gcc優化
- 一般來說，我們使用組合語言寫程式，就代表要直接對CPU或硬體做操作，甚至連指令的前後順序也不能對調（在『同步』章節會說明）
  - 例如：『flag=1; turn=0;』和『turn=0; flag=1;』雖然程式碼的執行順序不一樣，但對gcc而言是一樣的
  - gcc只保證『執行結果如這個程式單獨執行的結果』在不違反前述條件下，gcc會做「指令排程」

自己動手做

- 請設計一個實驗測試 『__asm__ volatile』和『 __asm__ 』的差別
  - 首先應該要打開gcc的優化參數
  - 故意寫出可以被優化的程式碼

caller-callee convention (素養)

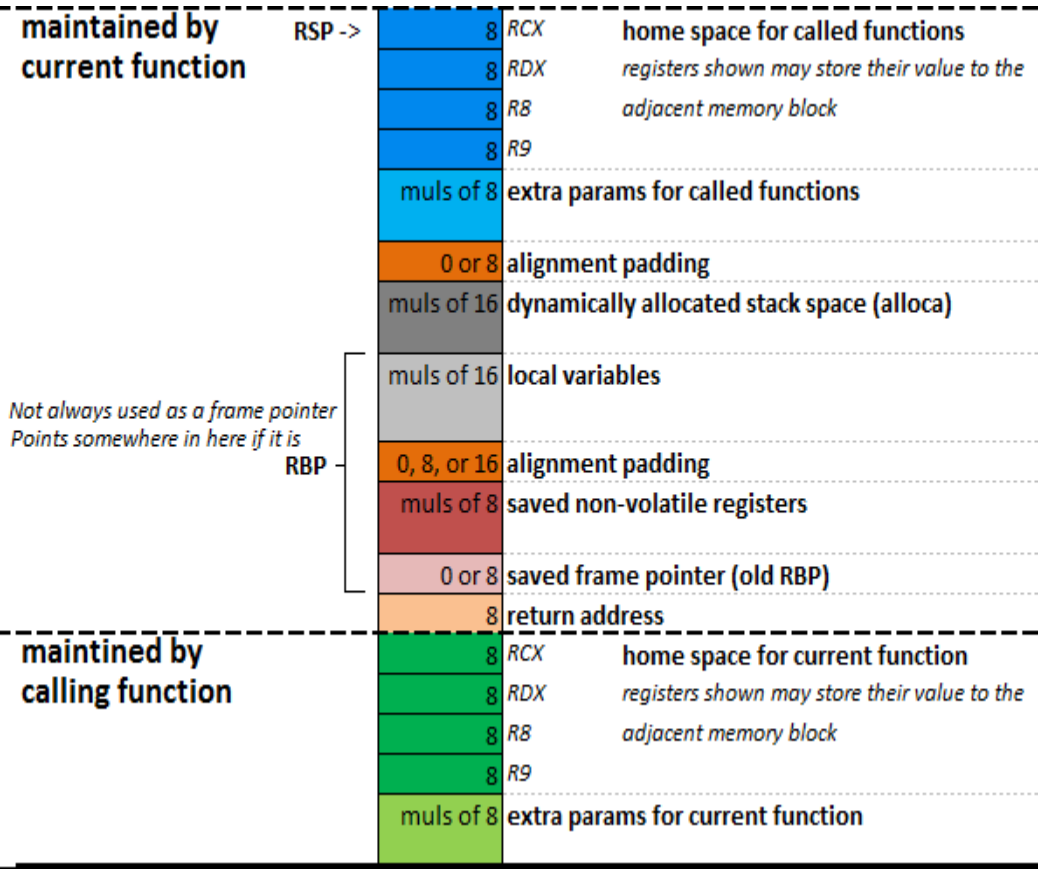
Figure 3.4: Register Usage

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of SSE registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0	temporary register; used to return long double arguments	No
%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system use (as thread specific data register)	No

- Caller-save：呼叫者負責儲存
  - 如果呼叫者等一下需要用到原本的值
- Callee-save：被呼叫者負責儲存
  - 如果被呼叫者改變了這個值

https://stackoverflow.com/questions/18024672/what-registers-are-preserved-through-a-linux-x86-64-function-call

呼叫堆疊（素養）



- 基於caller-callee convention就可以產生左方的堆疊
- 編譯器為了速度可能會有額外的padding
- 為了實現alloca，堆疊上可能有額外空間
- 注意：左圖是Windows的，但Linux也差不多是這樣

https://www.gamasutra.com/view/news/178446/Indepth_Windows_x64_ABI_Stack_frames.php

第二個範例

- 注意事項：當資料的「型別」不一樣時，必須使用不同的組合語言
- 與C語言不同的是：資料的型別是「編碼在組合語言中」
  - C會自動依照型別產生正確的組合語言

圖片來源：https://www.hiclipart.com/free-transparent-background-png-clipart-dwxzu

asm.2.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      long int a=10;
4.      long int b=20;
5.      long int c=30;
6.      long int d=40;
7.      __asm__ volatile (
8.                  "mov $100, %%rax\n"
9.                  "mov $200, %%rbx\n"
10.                 "add %%rbx, %%rax\n"
11.                 "mov %%rax, %0\n"
12.                 : "=m" (b)    //output
13.                 : "g" (a), "g" (d) //input
14.                 : "rbx", "rax" //搞爛掉的暫存器
15.                 );
16.     printf("a=%ld, b=%ld, c=%ld d=%ld\n", a, b, c, d);
17. }
```

輸出結果

```
ubuntu@oslab:~/os-lab/sharedFolder$ ./asm.2
a=10, b=300, c=30 d=40
```

自己動手做

- 試試看，可不可以把mov換成movl?
- 試試看，可不可以把左邊程式碼中『~~刪掉的部分~~』，拿掉可否跑?

```
1.  __asm__ volatile (
2.      "mov $100, %%rax\n"
3.      "mov $200, %%rbx\n"
4.      "add %%rbx, %%rax\n"
5.      "mov %%rax, %0\n"
6.      : "=m" (b)    //output
7.      : "g" (a), "g" (d) //input
8.      : ~~"rbx", "rax"~~ //搞爛掉的暫存器
9.  );
```

作業系統概論基於GNU/Linux

中正大學，資工系，作業系統實驗室，副教授 羅習五，shiwulo@gmail.com

asm.3.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      int a=10, b=20, c=90, d=100;
4.      printf("a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
5.      __asm__ volatile (
6.              "mov %1, %%rax\n"   //  rax = a
7.              "mov %2, %%rbx\n"   //  rbx = b
8.              "add %%rbx, %%rax\n"   // rax += rbx
9.              "mov %%rax, %0\n"   // c = rax
10.             : "=m" (c)  //output, %0
11.             : "g" (a), "g" (b) //input, %1, %2
12.             : "rbx", "rax" //搞爛掉的暫存器
13.             );
14.     printf("c = a + b\n");
15.     printf("a=%d, b=%d, c=%d, d=%d\n", a, b, c, d);
16. }
```

輸出結果

- a=10, b=20, c=90, d=100
- c = a + b
- a=120, b=20, c=30, d=100
- 結果錯誤，請修正這一個錯誤
  - 型別。。。

作業系統概論基於GNU/Linux

中正大學，資工系，作業系統實驗室，副教授 羅習五，shiwulo@gmail.com

asm.4.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      unsigned long msr;
4.      asm volatile ( "rdtsc\n\t"  // Returns the time in EDX:EAX.
5.                  "shl $32, %%rdx\n\t"    // Shift the upper bits left.
6.                  "or %%rdx, %0"          // 'Or' in the lower bits.
7.                  : "=a" (msr)          //msr會放在rax暫存器
8.                  :
9.                  : "rdx");
10.     printf("msr: %lx\n", msr);
11. }
```

編譯器可能會直接用RAX完全取代MSR

```c
#include <stdio.h>
int main(int argc, char** argv) {
    unsigned long msr;
    asm volatile ( "rdtsc\n\t"  // Returns the time in EDX:EAX.
        "shl $32, %%rdx\n\t"    // Shift the upper bits left.
        "or %%rdx, %0"          // 'Or' in the lower bits.
        : "=a" (msr)            //msr會放在rax暫存器
        :
        : "rdx");
    return (int)msr;
}
```

Source code

```
SHELL = /bin/bash
CC = gcc
CFLAGS = -g -O3 --static
SRC = $(wildcard *.c)
EXE = $(patsubst %.c, %, $(SRC))

all: ${EXE}

%:      %.c
        ${CC} ${CFLAGS} $@.c -o $@

clean:
        rm ${EXE}
```

makefile

反組譯

```
(gdb) disass /m main
Dump of assembler code for function main:
3       int main(int argc, char** argv) {

4           unsigned long msr;
5           asm volatile ( "rdtsc\n\t"  // Returns the time in EDX:EAX.
   0x00000000004015f0 <+0>:     rdtsc
   0x00000000004015f2 <+2>:     shl    $0x20,%rdx
   0x00000000004015f6 <+6>:     or     %rdx,%rax

6               "shl $32, %%rdx\n\t"    // Shift the upper bits left.
7               "or %%rdx, %0"          // 'Or' in the lower bits.
8               : "=a" (msr)            //msr會放在rax暫存器
9               :
10              : "rdx");
11          return (int)msr;
12      }
   0x00000000004015f9 <+9>:     retq

End of assembler dump.
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004015f0 <+0>:     rdtsc
   0x00000000004015f2 <+2>:     shl    $0x20,%rdx
   0x00000000004015f6 <+6>:     or     %rdx,%rax
   0x00000000004015f9 <+9>:     retq
End of assembler dump.
(gdb) q
```

編譯器可能會直接用RAX完全取代MSR

```c
#include <stdio.h>
int main(int argc, char** argv) {
    unsigned long msr;
    asm volatile ( "rdtsc\n\t"   // Returns the time in EDX:EAX.
        "shl $32, %%rdx\n\t"     // Shift the upper bits left.
        "or %%rdx, %0"           // 'Or' in the lower bits.
        : "=a" (msr)             //msr會放在rax暫存器
        :
        : "rdx");
    return (int)msr;
}
```

```
SHELL = /bin/bash
CC = gcc
CFLAGS = -g -O3 --static
SRC = $(wildcard *.c)
EXE = $(patsubst %.c, %, $(SRC))

all: ${EXE}

%:      %.c
        ${CC} ${CFLAGS} $@.c -o $@

clean:
        rm ${EXE}
```

結論：
因為“=a”(MSR)可能讓gcc將MSR配置在RAX暫存器，因此這個暫存器會繼續用。
如果把RAX定義成「髒掉的」與編譯器的優化不合

```
(gdb) disass /m main
Dump of assembler code for function main:
3       int main(int argc, char** argv) {

4           unsigned long msr;
5           asm volatile ( "rdtsc\n\t"  // Returns the time in EDX:EAX.
   0x00000000004015f0 <+0>:     rdtsc
   0x00000000004015f2 <+2>:     shl    $0x20,%rdx
   0x00000000004015f6 <+6>:     or     %rdx,%rax

6               "shl $32, %%rdx\n\t"    // Shift the upper bits left.
7               "or %%rdx, %0"          // 'Or' in the lower bits.
8               : "=a" (msr)            //msr會放在rax暫存器
9               :
10              : "rdx");
11          return (int)msr;
12      }
   0x00000000004015f9 <+9>:     retq

End of assembler dump.
(gdb) disass main
Dump of assembler code for function main:
   0x00000000004015f0 <+0>:     rdtsc
   0x00000000004015f2 <+2>:     shl    $0x20,%rdx
   0x00000000004015f6 <+6>:     or     %rdx,%rax
   0x00000000004015f9 <+9>:     retq
End of assembler dump.
(gdb) q
```

輸出結果

```
ubuntu@oslab:~/os-lab/sharedFolder$ ./asm.4
msr: 1798fd4d0135
```

自己動手做

- 目的：實際上clock_gettime內部呼叫了RDTSCP，因此其結果會差不多，但作業系統可能會做一些校正。
- 比較一下量出來的時間與clock_gettime()（POSIX）是否有差異?
- 如果是計算「差值」是否會一樣?

素養

- 練習看一下組合語言的文件
- 將rdtsc改成rdtscp會有什麼樣的不一樣?
- cat /proc/cpuinfo | grep tsc會出現相關的『ＣＰＵ技術』
  - tsc、rdtsc、rdtscp、constant_tsc、nonstop_tsc、tsc_deadline_timer、tsc_adjust
  - 上述技術背後的涵義為何?
- ARM、MIPS、RISC-V也有相類似的技術

asm.5.c

```
1.  #include <stdio.h>
2.  int main(int argc, char** argv) {
3.      unsigned long msr;
4.      asm volatile ( "rdtsc\n\t"  // Returns the time in EDX:EAX.
5.                  "shl $32, %%rdx\n\t"    // Shift the upper bits left.
6.                  "or %%rdx, %0"          // 'Or' in the lower bits.
7.                  : "=m" (msr)            //msr會放在記憶體
8.                  :
9.                  : "rdx");
10.     printf("msr: %lx\n", msr);
11. }
```

輸出結果

- ubuntu@oslab:~/os-lab/sharedFolder$ ./asm.5
- msr: 184300000000
- 結果錯誤，請修正這一個錯誤

進階文件

- 如果想要學好inline assembly，請閱讀官方文件
  - https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
- 或者，大概看過，知道有哪些功能，需要的時候再去查文件

小結論

- 我們之前的課程學過組合語言
- 但在Linux中除了單獨的組合語言檔案以外，還有「嵌入在C語言」的組合語言，我們必須很熟悉
- 某些組合語言具有特殊的功能，通常與作業系統相關
  - syscall：發出system call
  - cr3暫存器的所有相關指令：控制『虛擬記憶體』
  - 未來會用到這些指令

作業系統概論基於GNU/Linux

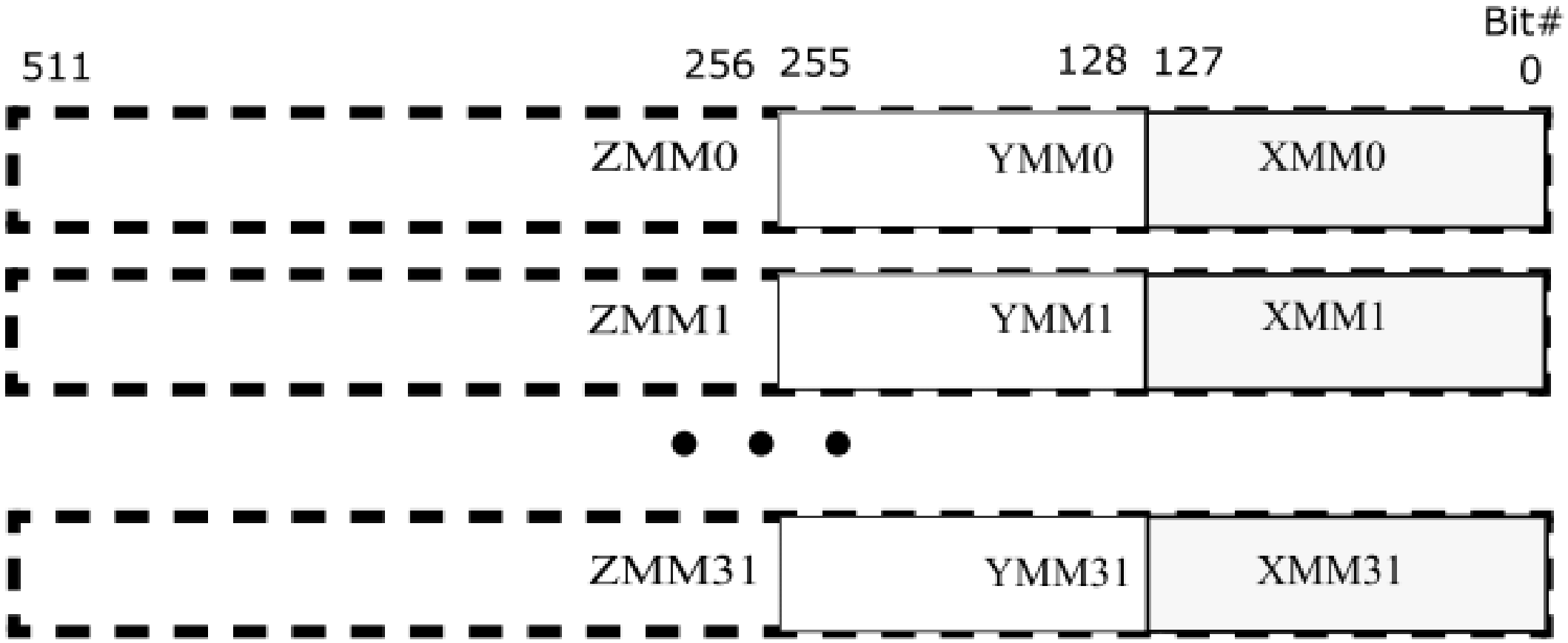
中正大學，資工系，作業系統實驗室，副教授 羅習五，shiwulo@gmail.com

環境設定（非必須）

- 下載『最新版本的』gdb的source code
  - https://www.gnu.org/software/gdb/download/
- 在電腦上安裝python3.8及python3.8-dev
- 使用sudo update-alternatives將python3.8設為預設的python
- 設定gdb成你想要的樣子
  - https://gef.readthedocs.io/en/latest/config/
  - https://gef.readthedocs.io/en/latest/commands/context/

圖片來源：https://www.hiclipart.com/free-transparent-background-png-clipart-dwxzu

Intel 浮點運算


Bit#
511
256
255
128
127
0
ZMM0
YMM0
XMM0
ZMM1
YMM1
XMM1
ZMM31
YMM31
XMM31

Intel的系統暫存器

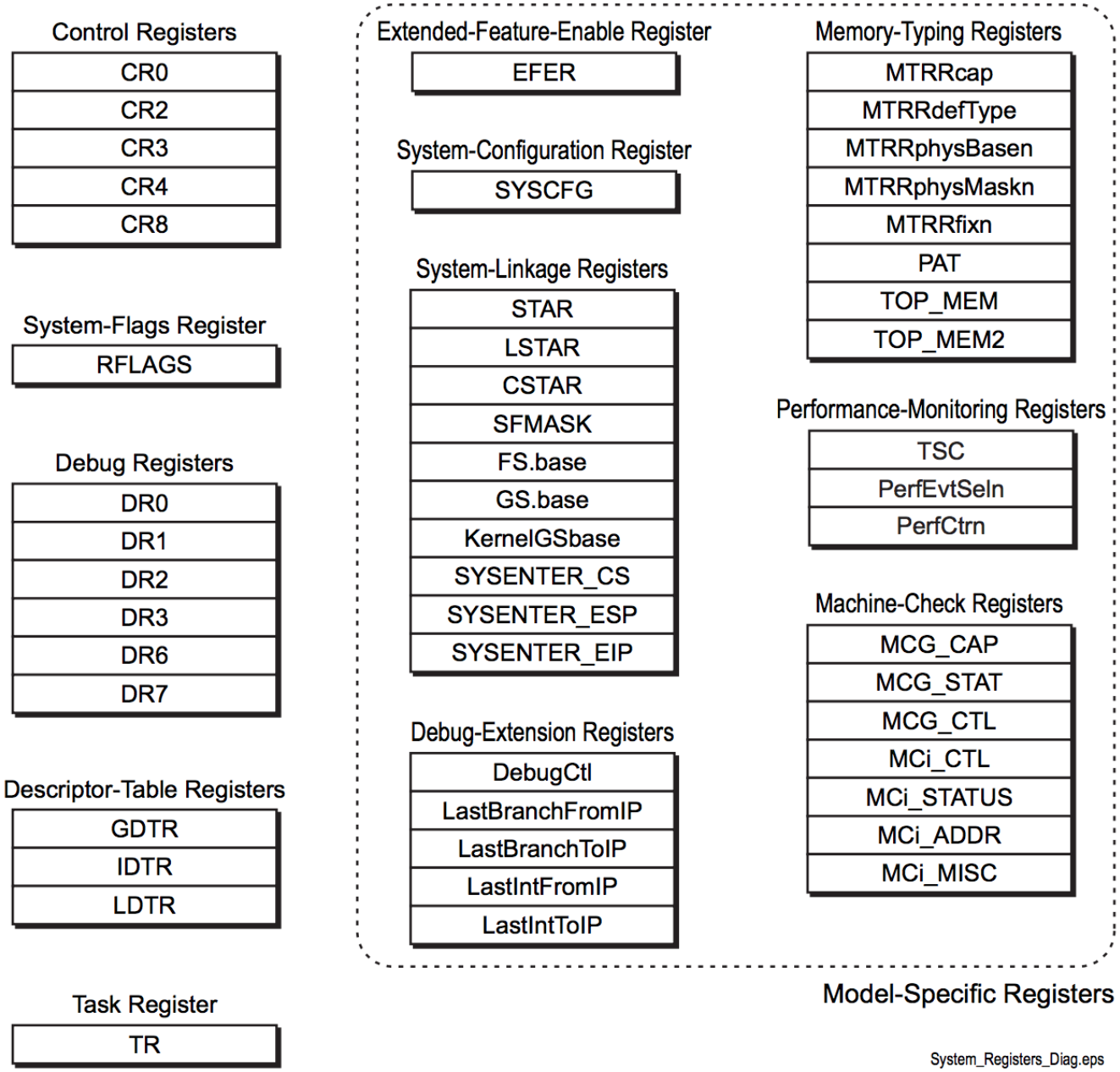


Figure 1-7. System Registers

- 大概介紹86的語法
  - https://software.intel.com/content/www/us/en/develop/articles/introduction-to-x64-assembly.html?wapkw=