Chapter 2

Procedure Call and Other Architectures

Outline

- Supporting Procedures in Computer Hardware
- Communicating with People
- RISC-V Addressing for Wide Immediates and Addresses
- Parallelism and Instructions: Synchronization
- Translating and Starting a program
- A C Sort Example to Put it All Together
- Arrays versus Pointers
- Real Stuff: MIPS Instructions
- Real Stuff: x86 Instructions
- Real Stuff: The Rest of the RISC-V Instruction Set
- Fallacies and Pitfalls
- Concluding Remarks

Procedure Call: An Example

```
#include <stdio.h>
#include <stdint.h>
int64_t sum_array(int64_t nums[], int size){
    int64_t sum=0;
    for (int i = 0; i < size; ++i){
        sum=sum+nums[i];
    }
    return sum;
}

int main(){
    int size = 4;
    int64_t list[] = {3, 5, 4, 6};
    int64_t output = 0;

    output=sum_array(list, size);
    printf("Sum=%lld\n", output);
    return 0;
}
```

`int64_t == long long int` (8 bytes)

Example in Memory

程式Compile過後的Instructions會以32bits存放在Memory Space裡面，並以label(Ex: .main、.sum_array)去做分塊

Assume we compile the program and put the binary codes in memory

```
int64_t sum_array(int64_t nums[], int size){
    int64_t sum=0;
    for (int i = 0; i < size; ++i){
        sum=sum+nums[i];
    }
    return sum;
}
```

結束之後，Program Counter會跳回去原本運行到的位置

```
int main(){
    int size = 4;
    int64_t list[] = {3, 5, 4, 6};
    int64_t output = 0;

    output=sum_array(list, size);
    printf("Sum=%lld\n", output);

    return 0;
}
```

做Funciton Call時，Program Counter會先跳到sum_array函式的位置

Memory space

Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
     規定的function call存放parameter的位置
     可不遵守，但不符合ABI格式可能不相容
  2. Transfer control to procedure
  3. Acquire storage for procedure 得到local variable的記憶體空間
  4. Perform procedure's operations
  5. Place result in register for caller (x10, x11)
  6. Return to place of call (address in x1)
     回到呼叫函式時的位置

Procedure Call Instructions

- Procedure call: jump and link

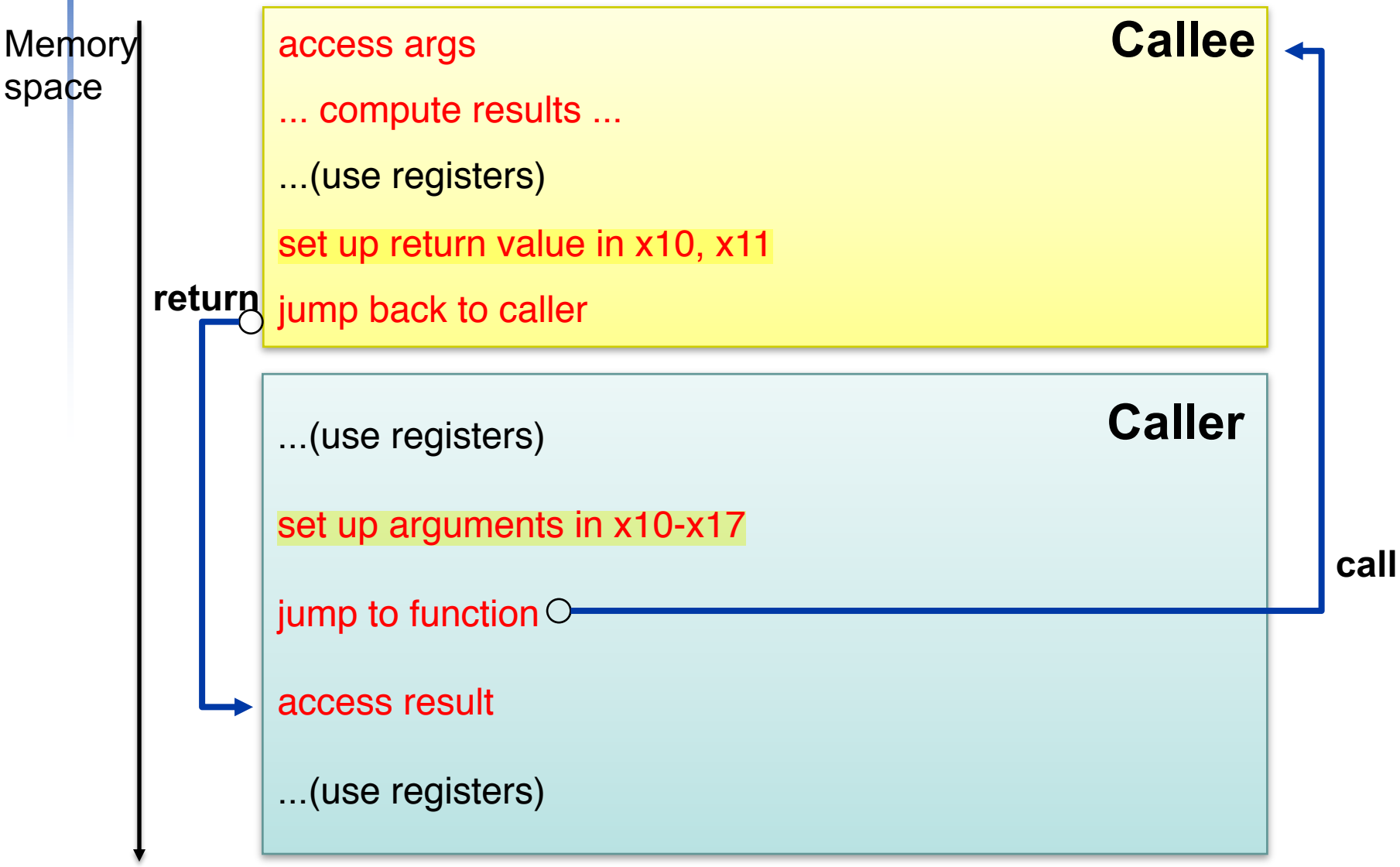
  `jal x1, ProcedureLabel` 同beq x0, x0, L1 但jal可以做到把該行下一個要做的Instruction的位置存到x1，讓之後可以跳回來(非Label的位置)

  - Address of following instruction put in x1
  - Jumps to target address
- Procedure return: jump and link register

  `jalr x0, 0(x1)` Label做完後，Return到剛剛記住的x1的位置繼續做。和jal一樣會記錄下一個Insturction的位置，但不重要所以存至x0 當link address存在register裡時用jalr、Label時則用jal

  - Jumps to 0 + address in x1
  - Use x0 as rd
    - We do not need the return address and x0 cannot be changed (always 0)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

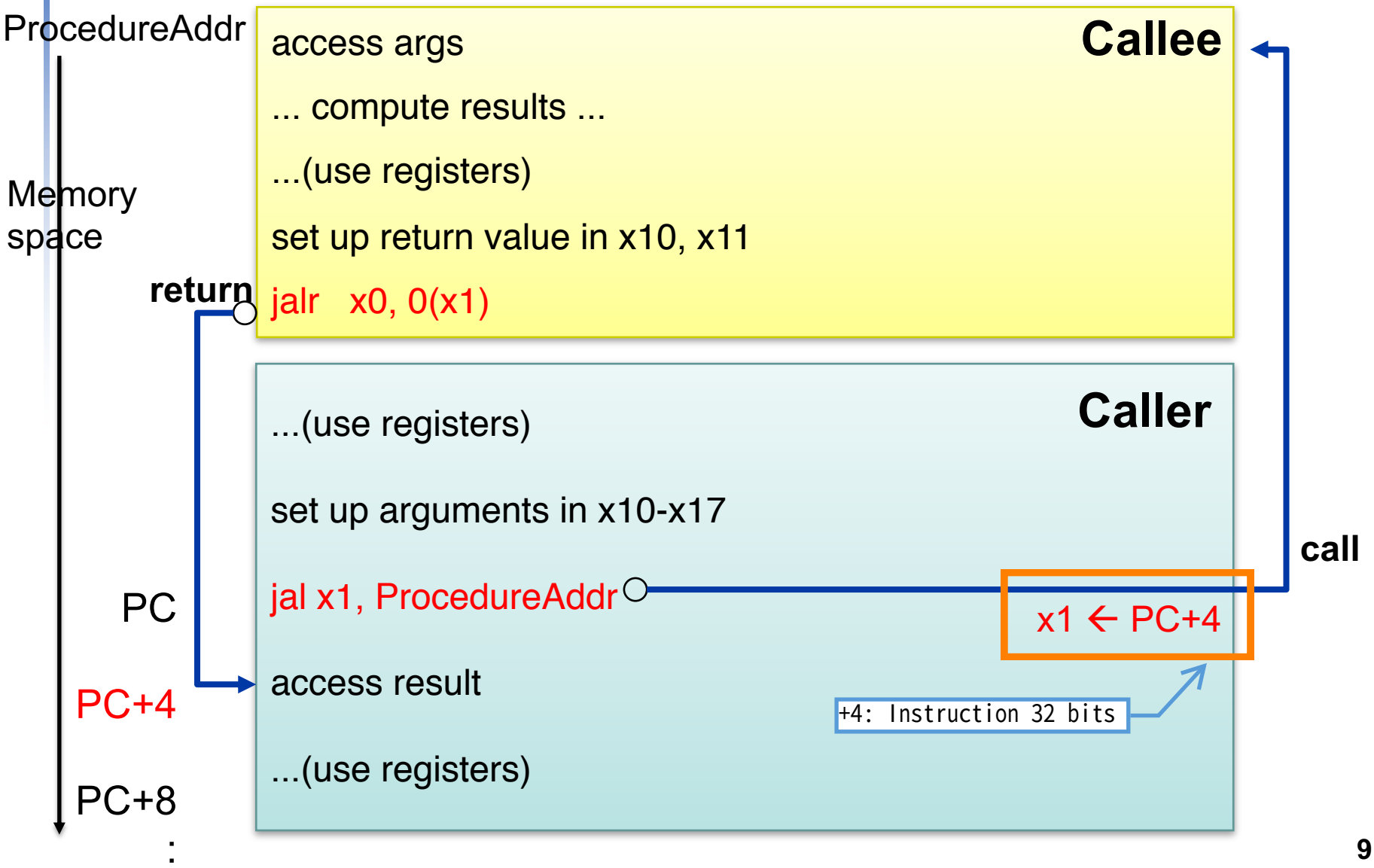
Procedure Call Assembly Steps



Execution of a Procedure

1. Place parameters in a place where the procedure can access them (via x10 – x17)
   - If more than 8 parameters, push them into the beginning of stack frame. More later.
2. Transfer control to the procedure by jal x1, ProcedureAddress
3. Acquire the storage resources needed for the procedure
4. Perform the desired task
5. Place the result value in a place where the calling program can access it (via x10, x11)
6. Return control to the point of origin by jalr x0, 0(x1)

Procedure Call with jal



Leaf Procedure Example

不會呼叫別的函式(在尾端)

- C code:

```
int64_t leaf_example (int64_t g, int64_t h,
int64_t i, int64_t j) {
      int64_t f;
      f = (g + h) - (i + j);
      return f;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x20 on stack (x20 is a saved register)

(g+h)->x5 (i+j)->x6

Temporary Register: 可以隨意使用，呼叫者不會去用它
Saved Register:呼叫者(Caller)會使用到的register，被呼叫者(Callee)必須先存在sp之後，並於使用完後load回去。

RISC-V code of Example

```
leaf_example:
        addi sp,sp,-8 //Save x20 on stack
        sd   x20,0(sp)
        add  x5,x10,x11 //x5 = g + h
        add  x6,x12,x13 //x6 = i + j
        sub  x20,x5,x6 //f = x5 - x6
        addi x10,x20,0 //copy f to return register
        ld   x20,0(sp) //Restore x20 from stack
        addi sp,sp,8
        jalr x0,0(x1) //Return to caller
```

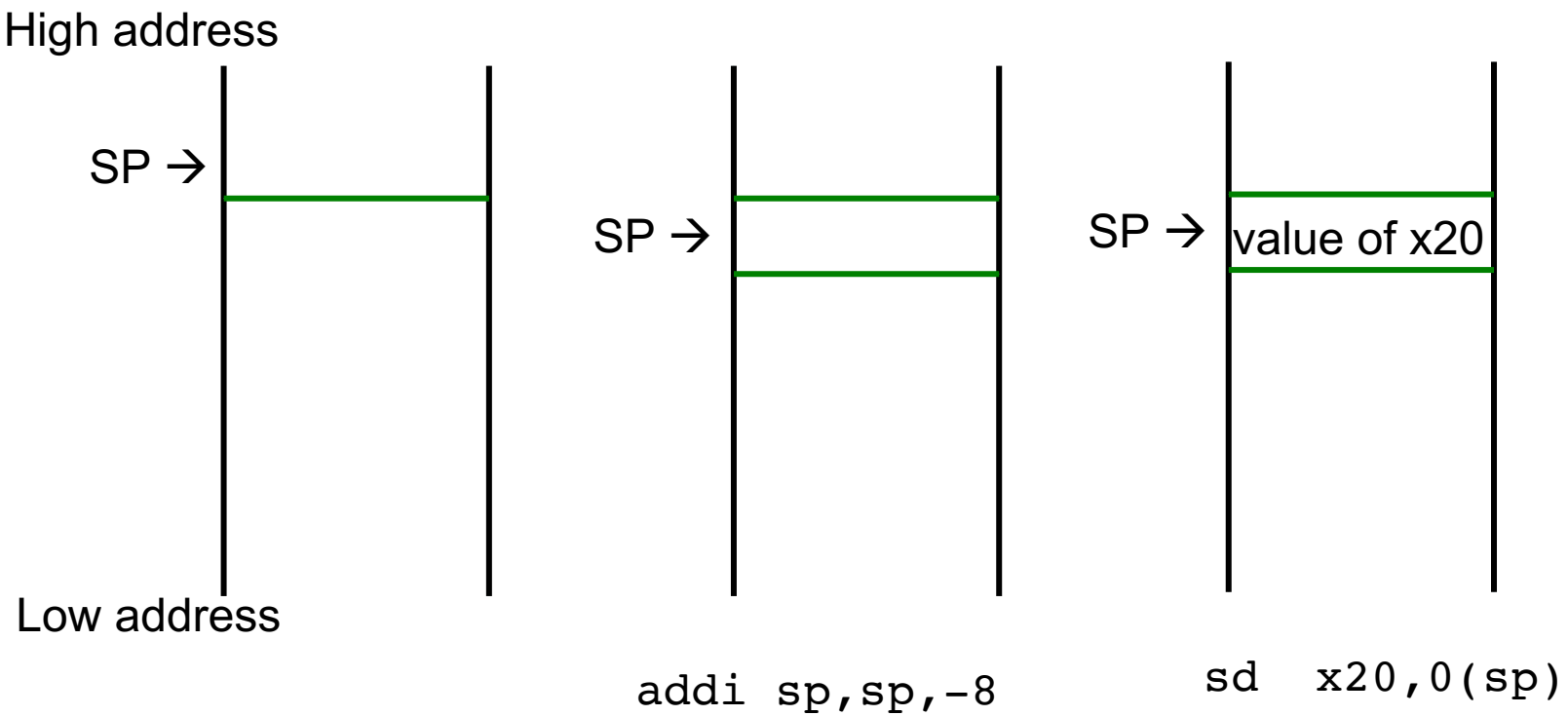
記憶體往下長

向Stack Pointer借空間，並把x20暫存在其中

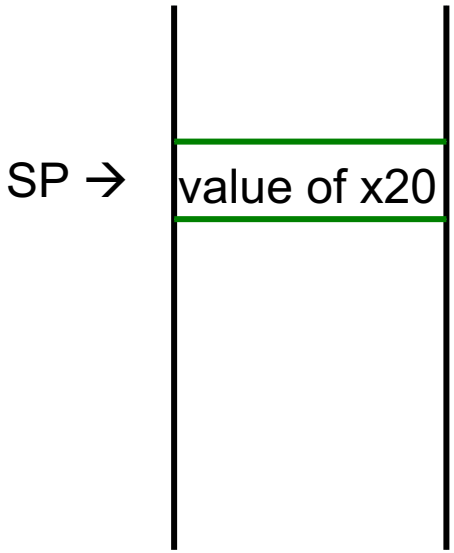
前面x20被更改過，所以從Stack Pointer裡面load回來

把Stack Pointer減回去，釋放該空間

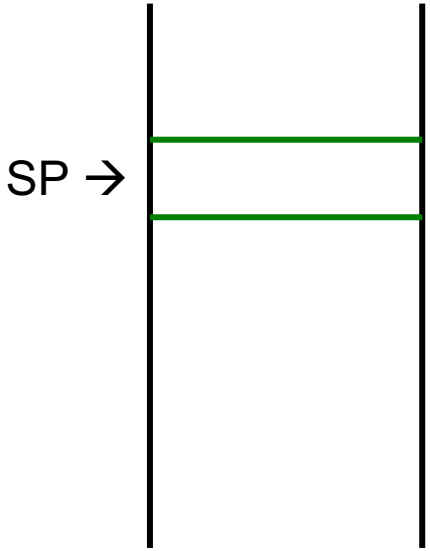
Save Registers on Stack (Push)



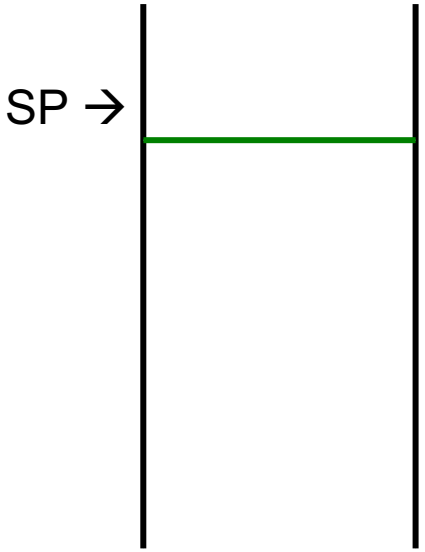
Restore Registers on Stack (Pop)




```
ld x20,0(sp)
```




```
addi sp,sp,8
```



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its **return address**
  - Any **arguments and temporaries** needed after the call
- Restore from the stack after the call

Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee Callee隨便用

- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Callee須保證其正確性

RISC-V Register Names

Register	Assembly name	Description	Saver
x0	zero	Hard-wired zero	–
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	–
x4	tp	Thread pointer	–
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

若main()->f1()->f2()
當f1呼叫f2時，會複寫x1的值。
所以f1要把main呼叫它時x1先存起來
等f2執行完後再load回來，故Saver
是Caller

被呼叫的人Callee要保證Stack
Pointer裡的內容不會被更動

當f1呼叫f2時，若要使f2執行完之後x5值不變，
f1(Caller)就要先把x5存在sp裡面(保護)

當f1呼叫f2時會傳入參數，可能會更動
到main給f1的參數，所以f1(Caller)要
先將x10存起來

當f1呼叫f2時，若f2要用到x18時，它就
要先把x18先存進sp，用完之後再pop
出來存回x18，確保其不被更動

Non-Leaf Procedure Example

會呼叫別人(不在尾端)

- C code:

```
int64_t fact (int64_t n){
     if (n < 1) return 1;
     else return n * fact(n - 1);
}
```

  - Argument n in x10
  - Result in x10

RISC-V code of Example

```
fact:
        addi sp,sp,-16
        sd   x1,8(sp)   //Save return address and n on stack
        sd   x10,0(sp)  //x10 = n
        addi x5,x10,-1  //x5 = n-1
        bge  x5,x0,L1   //if (x5>0), go to L1
        addi x10,x0,1   //else return x10 = 1
        addi sp,sp,16   //pop back and discard all stack values
        jalr x0,0(x1)   //return to (x1)
L1:     addi x10,x10,-1 //n = n-1
        jal  x1,fact    //call fact(n-1)
        addi x6,x10,0   //move return of fact(n-1) in x10 to x6
        ld   x10,0(sp)  //restore caller's n to x10
        ld   x1,8(sp)   //restore caller's return address to x1
        addi sp,sp,16   //pop back stack address
        mul  x10,x10,x6 //return n * fact(n-1)
        jalr x0,0(x1)   //return
```

要保留x1在最後return時使用，也要存在sp

fact每呼叫一次，sp都會變深

因為要保留x10在最後乘法的時候使用，要存在sp

紅線以上就是把n,n-1,n-2...1存到sp
順便存x1是為了之後反覆相乘運算操作方便

遞迴回到fact

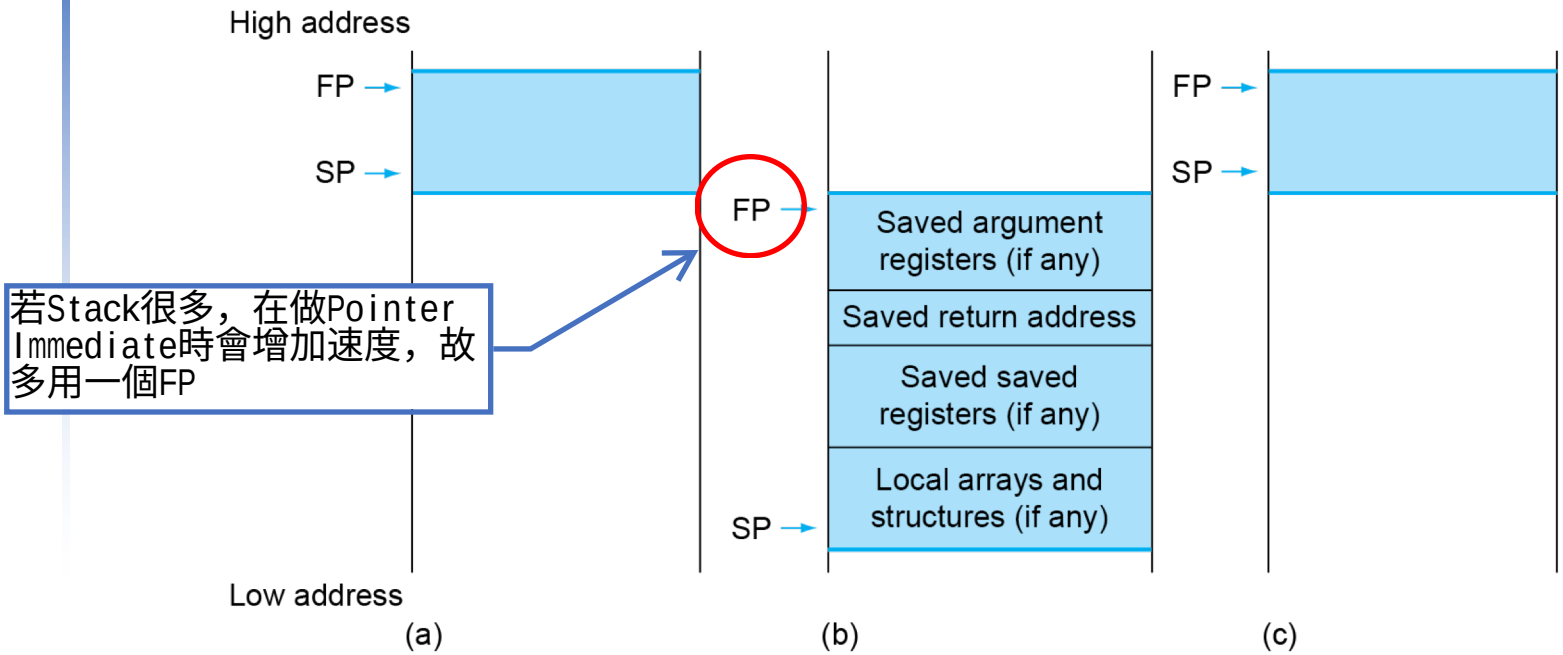
Issue 1: Register Conflict

- Caller and callee both use the same registers
- One example solution
  - Caller saves temporary registers (x5-x7, x28-x31) into a memory stack if they are to be used later
  - Callee saves saved registers (x8-x9, x18-x27) if it uses them
  - Both share the task of saving to memory

Issue 2: Need More Arguments

- Caller need to pass more arguments than 8 (x10, ..., x17) to callee
- Solution
  - Place extra variables and extra arguments onto stack (a LIFO queue)
  - x2 (stack pointer) points to most recently allocated address

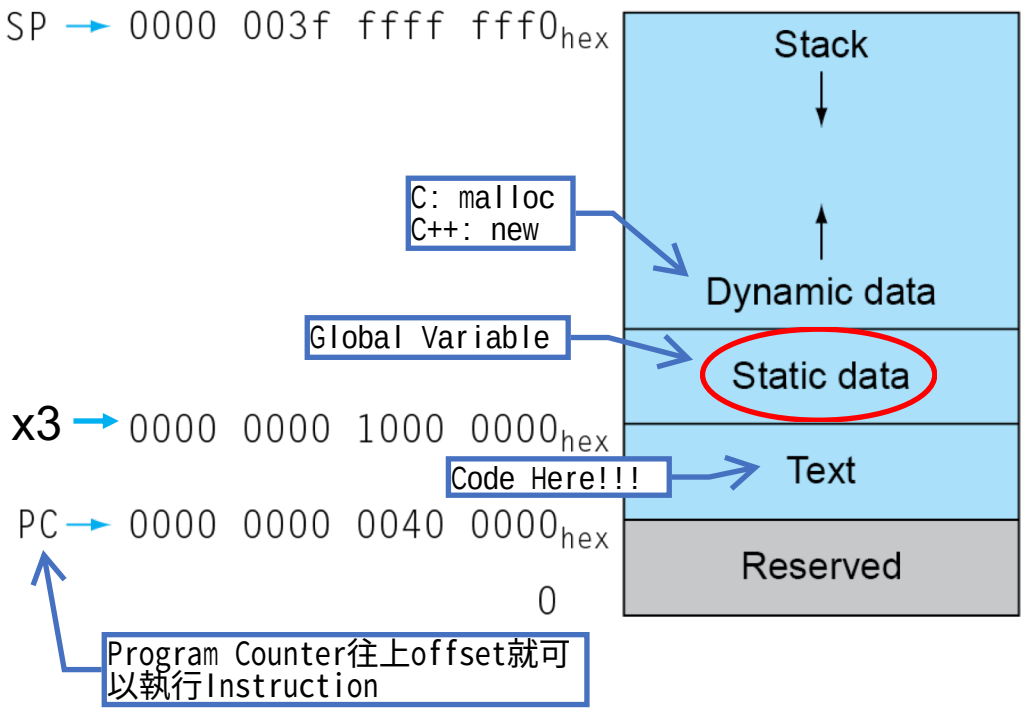
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables 在進行funtion call時，function內的local variable要存在sp內
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
  - May use x8 for FP

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{
    size_t i;
    i = 0;
    while ((x[i]=y[i])!='\0')
        i += 1;
}
```

RISC-V code of strcpy Example

```
strcpy:
        addi sp,sp,-8        // adjust stack for 1 doubleword
        sd   x19,0(sp)       // push x19
        add  x19,x0,x0       // i=0
L1:     add  x5,x19,x10      // x5 = addr of y[i]
        lbu  x6,0(x5)        // x6 = y[i]
        add  x7,x19,x11      // x7 = addr of x[i]
        sb   x6,0(x7)        // x[i] = y[i]
        beq  x6,x0,L2        // if y[i] == 0 then exit
        addi x19,x19, 1      // i = i + 1
        jal  x0,L1           // next iteration of loop
L2:     ld   x19,0(sp)       // restore saved x19
        addi sp,sp,8         // pop 1 doubleword from stack
        jalr x0,0(x1)        // and return
```

String的character是用Byte-encoded的，只佔8 bits(剛好一個offset)

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rd, constant`
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

要達成寫出32-bit constant的效果:
(1)原本addi只能允許12bits的imm
(2)所以需要另外的20bits的指令
(3)lui rd, const可以達成寫入第12-31 bits的操作(Instruction 32bit減去opcode 7bit, rd 5bit剛好剩下20bits)

immediate只有20 bits的空間，所以要運用移位去達到32 bits的Constant

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

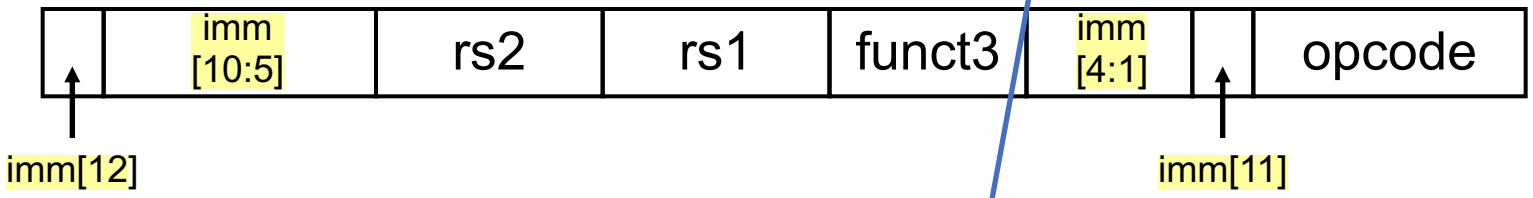
```
addi x19,x19,128  // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

1.一個Instruction通常為2或4個Bytes(即16bits或32bits)，因為都是以2為單位，所以LSB就不用存取，使用時再乘以2(Left shift 1)即可。
2.少存LSB，所以可以涵蓋的Target Address可以增倍。



- PC-relative addressing
  - Target address = PC + immediate × 2

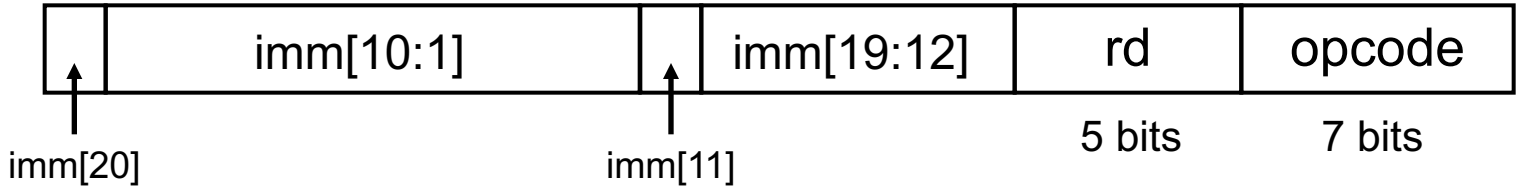
12 bits的範圍

若大於12bits的範圍(Offset不夠)，就先計算好Address存在某個Register後再Jump
Ex: lui x5,.... -> jalr x1, 0(x5)

Jump Addressing

不像Branch還需要做比較運算，所以imm的空間較大

- Jump and link (`jal`) target uses 20-bit immediate for larger range
- UJ format:



  - Target address = PC + immediate × 2
- For long jumps, eg, to 32-bit absolute address

大於20bit的範圍(ex:32bit)，就先計算好address(用lui和addi)，再使用jalr去jump

  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary

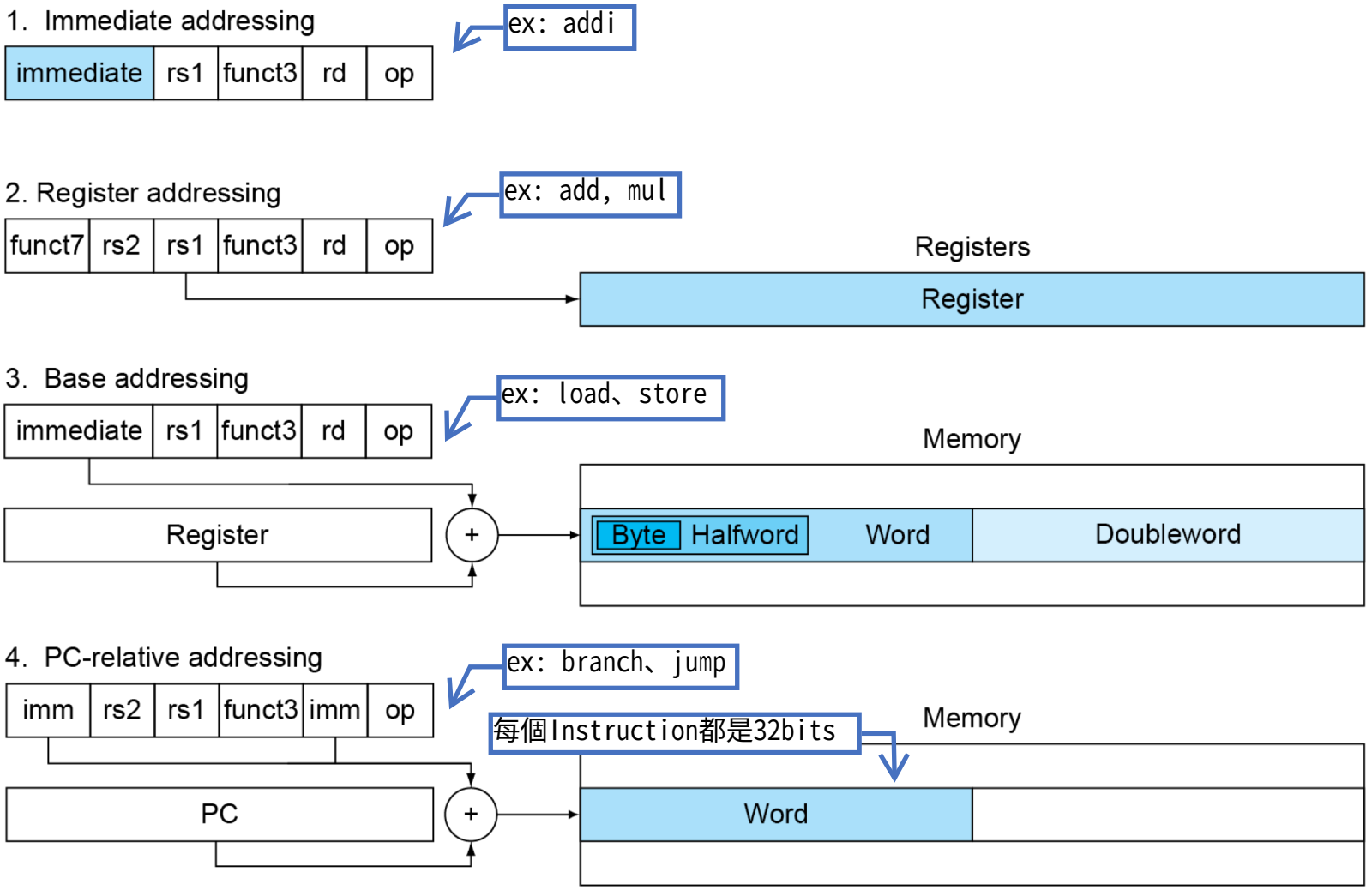


Table of RISC-V Instructions

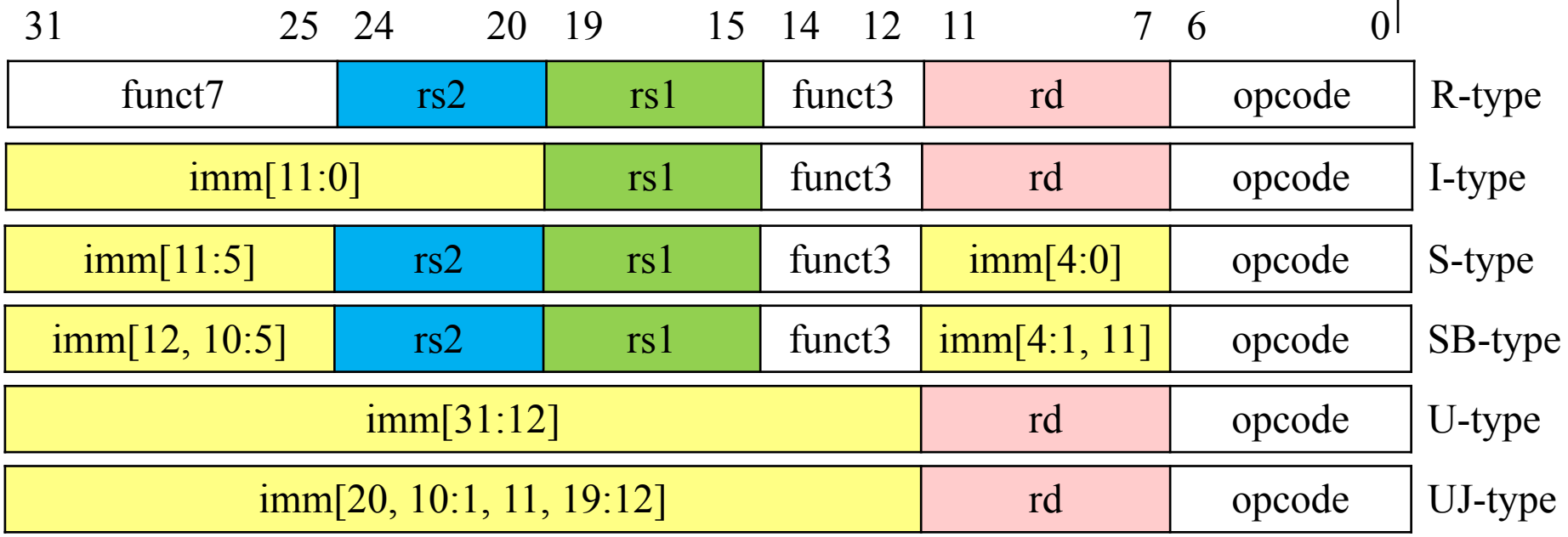
RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	x5 = x6 + x7	Three register operands
	Subtract	`sub x5, x6, x7`	x5 = x6 - x7	Three register operands
	Add immediate	`addi x5, x6, 20`	x5 = x6 + 20	Used to add constants
Data transfer	Load doubleword	`ld x5,40(x6)`	x5 = Memory[x6 + 40]	Doubleword from memory to register
	Store doubleword	`sd x5,40(x6)`	Memory[x6 + 40] = x5	Doubleword from register to memory
	Load word	`lw x5,40(x6)`	x5 = Memory[x6 + 40]	Word from memory to register
	Load word, unsigned	`lwu x5,40(x6)`	x5 = Memory[x6 + 40]	Unsigned word from memory to register
	Store word	`sw x5,40(x6)`	Memory[x6 + 40] = x5	Word from register to memory
	Load halfword	`lh x5,40(x6)`	x5 = Memory[x6 + 40]	Halfword from memory to register
	Load halfword, unsigned	`lhu x5,40(x6)`	x5 = Memory[x6 + 40]	Unsigned halfword from memory to register
	Store halfword	`sh x5,40(x6)`	Memory[x6 + 40] = x5	Halfword from register to memory
	Load byte	`lb x5,40(x6)`	x5 = Memory[x6 + 40]	Byte from memory to register
	Load byte, unsigned	`lbu x5,40(x6)`	x5 = Memory[x6 + 40]	Byte halfword from memory to register
	Store byte	`sb x5,40(x6)`	Memory[x6 + 40] = x5	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	x5 = Memory[x6]	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	Memory[x6] = x5; x7 = 0/1	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	x5 = 0x12345000	Loads 20-bit constant shifted left 12 bits

Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with consta
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate

Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greatr/eq, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

RISC-V Standard Base ISA Encoding

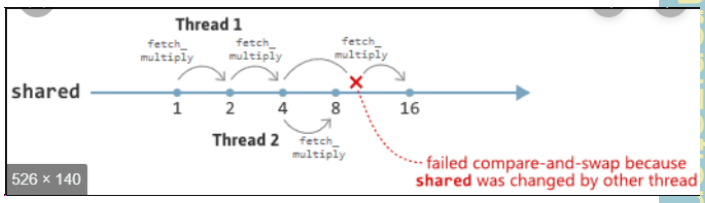


- R-type: arithmetic instructions
- I-type: Loads & immediate arithmetic
- S-type: Stores
- SB-type: Conditional branch format
- UJ-type: Unconditional jump format
- U-type: Upper immediate format

Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

存取讀取不同步，造成記憶體內容混淆



Thread2必須等Thread1做完之後才能做

Synchronization in RISC-V

- Load reserved: `lr.d rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.d rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.d`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

嘗試從rs2存值回去rs1，若發現原本的location已經改變(值被別人寫入)，rd會回傳非0，代表錯誤。

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:  lr.d x10,(x20)
        sc.d x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

load reserved in x20

存x23值進入x20

- Example 2: lock

要先等另一個unlock後才能去lock(possible sequences)

```
        addi x12,x0,1          // copy locked value
again:  lr.d x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.d x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

將1(x12)存回去x20，透過x11檢查是否成功(0?)

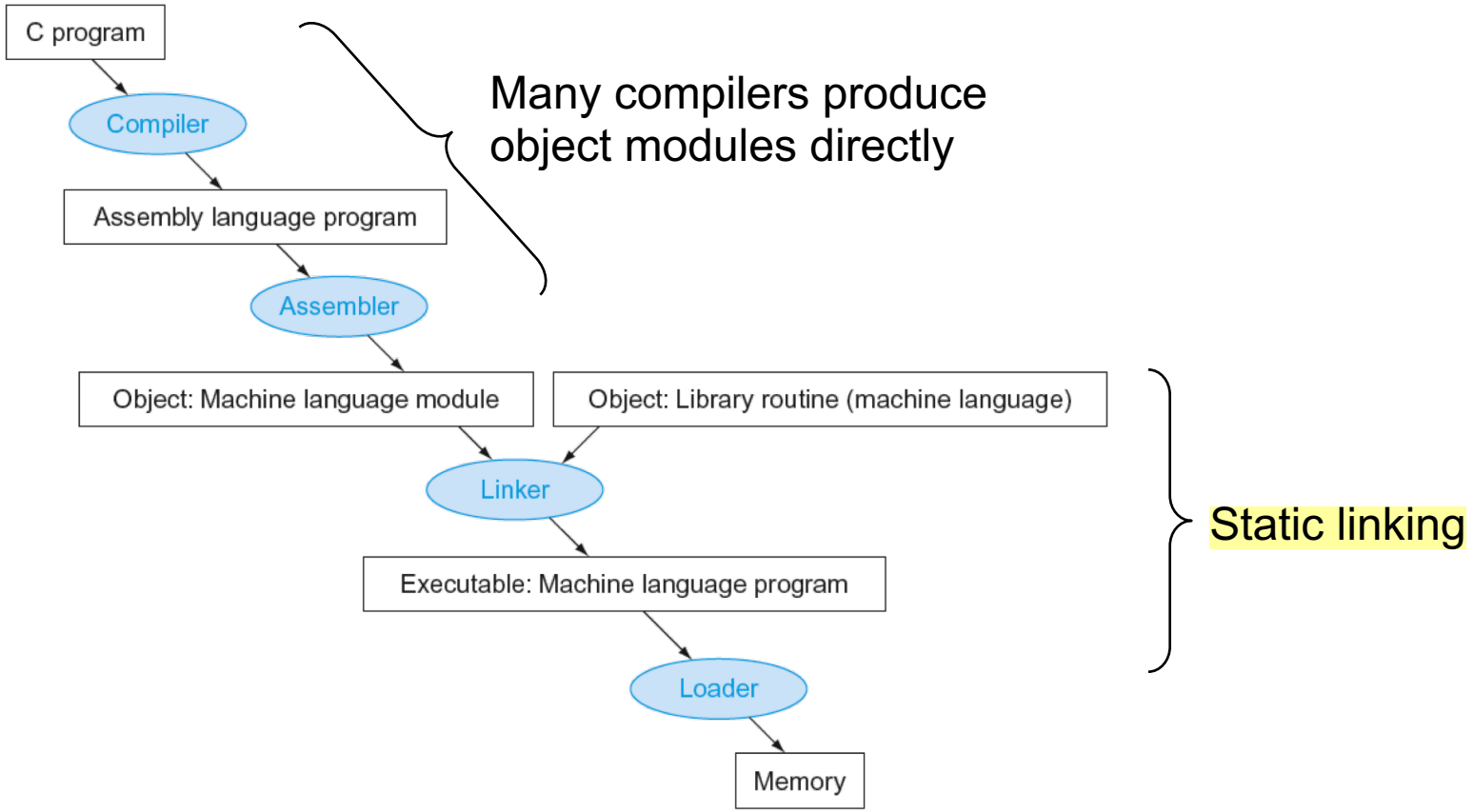
兩個bne(雙重把關):
(1)觀察目前Lock狀態
(2)若為free，搶著先做(透過store attempt)

- Unlock:

```
        sd   x0,0(x20)         // free lock
```

當x12要寫進去x20前，x20會檢查自己目前的值跟load reserve時的值有沒有改變(該值在lr.d指令時存在外部記憶體中)

Translation and Startup



Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

變數名稱(除錯用)

每個被定義的變量都應該要有一個獨一無二的名稱，引用的時候才不會跟其他變量混淆。我們將變量或函數通稱為符號 (symbol)，而他們的名稱則稱為符號名 (symbol name)。

Linking Object Modules

鏈結要做的事情，就是把多個不同的文件組合在一起，成為一個完整的整體。

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

等Library Module傳入時做Linking去指定絕對的位址(for jump)，下次呼叫函式時就可以知道絕對位址

在編譯期轉成組合語言時，我們已經知道代碼會呼叫哪些符號，但是我們並不知道這些符號真正的位址。當最後將代碼轉成機器語言時，我們會先賦予這些符號一個假的位址，並且用一個重定位表去記錄有哪些符號的位址是假的，之後需要再次修改。
而這些資訊就會被記錄在目標文件內的符號表 (symbol table) 以及重定位表 (relocation table) 中。
而在此階段，鏈結器 (linker) 便會分配符號所需的地址，並且依照符號表與重定位表來重新修改機器語言的內容。

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including sp, fp, gp)
  6. Jump to startup routine
     - Copies arguments to x10, … and calls main
     - When main returns, do exit syscall

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

使用動態連結機制時，函式庫可以先不需要被連結進來，而是在需要的時候才透過動態連結器 (Dynamic Linker) 尋找並連結函式庫，這種方式可以不用載入全部的程式，因此可以節省記憶體。還可以節省編譯、組譯、連結所花費的時間。

當程式第一次執行到動態函數時，動態連結器會搜尋看看該函數是否已經在記憶體中，如果有則會跳到該函數執行，如果沒有則會呼叫載入器，動態的將該函式庫載入到記憶體，然後才執行該函數。

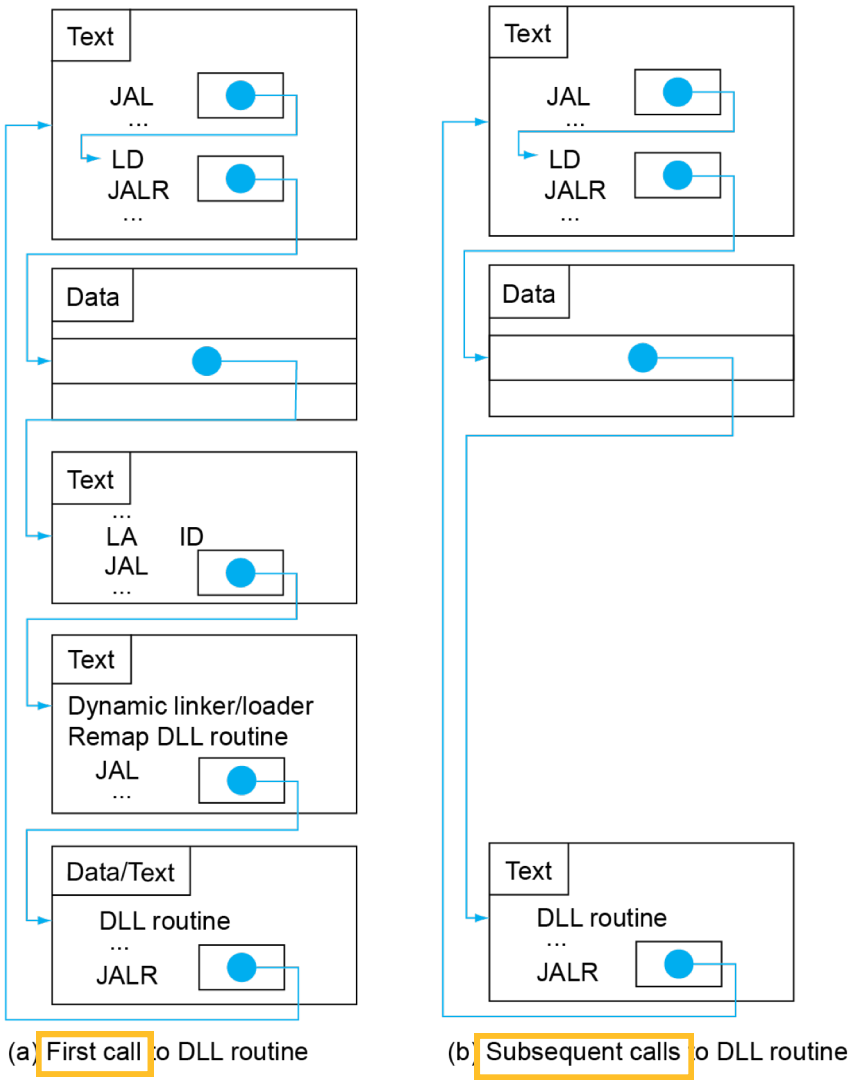
Lazy Linkage

Indirection table

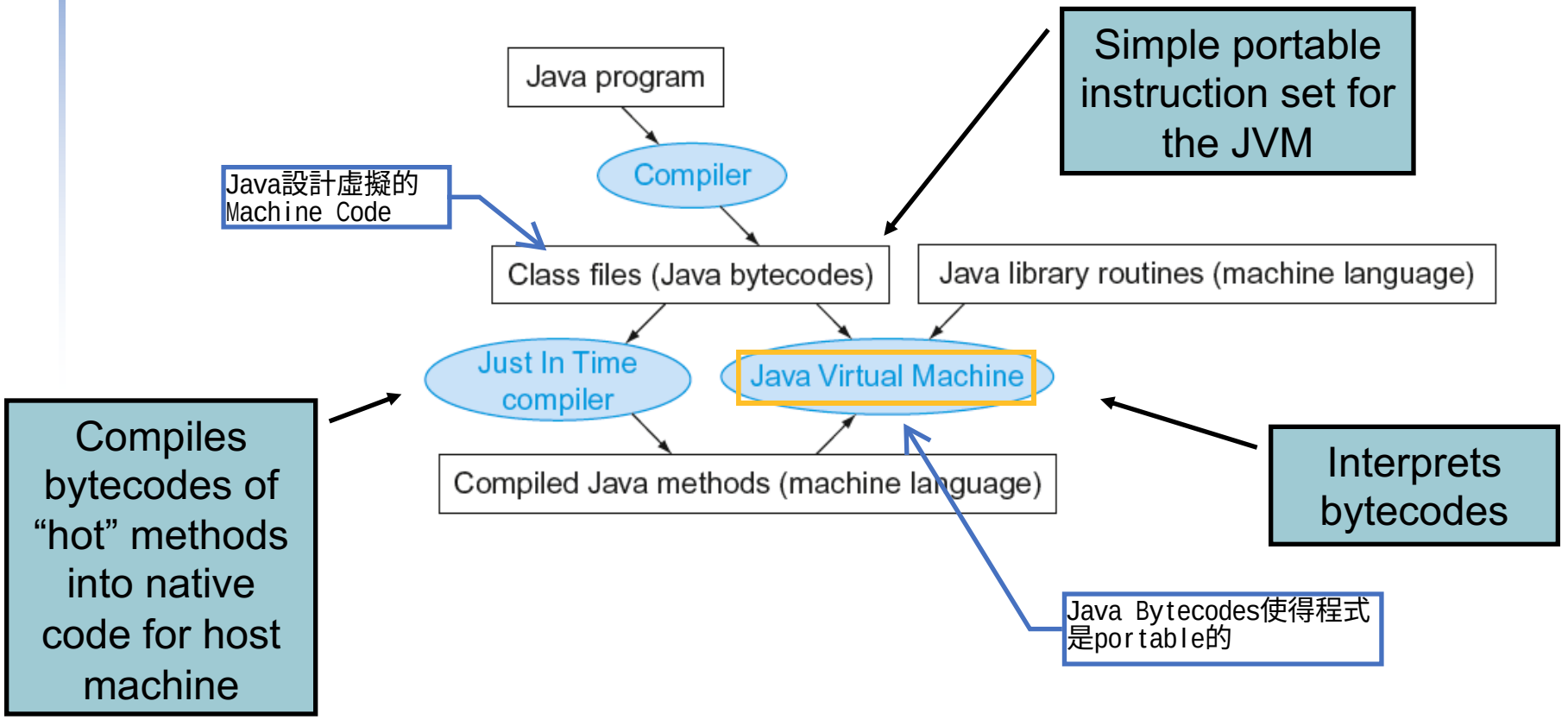
Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code



Starting Java Applications



C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int64_t v[], int64_t k)
{
  int64_t temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in x10, k in x11, temp in x5

The Procedure Swap

```
swap:
   slli x6,x11,3     // reg x6 = k * 8
   add  x6,x10,x6    // reg x6 = v + (k * 8)
   ld   x5,0(x6)     // reg x5 (temp) = v[k]
   ld   x7,8(x6)     // reg x7 = v[k + 1]
   sd   x7,0(x6)     // v[k] = reg x7
   sd   x5,8(x6)     // v[k+1] = reg x5 (temp)
   jalr x0,0(x1)     // return to calling routine
```

讀出來
做互換

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int64_t v[], size_t n){
  size_t i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
    li   x19,0              // i = 0
for1tst:
    bge  x19,x11,exit1   // go to exit1 if x19 ≥ x11 (i≥n)

    (body of outer for-loop)

    addi x19,x19,1       // i += 1
    j    for1tst         // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

v in x10, n in x11,
i in x19, j in x20

```
    addi x20,x19,-1     // j = i –1
for2tst:
    blt  x20,x0,exit2   // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,3       // reg x5 = j * 8
    add  x5,x10,x5      // reg x5 = v + (j * 8)
    ld   x6,0(x5)       // reg x6 = v[j]
    ld   x7,8(x5)       // reg x7 = v[j + 1]
    ble  x6,x7,exit2    // go to exit2 if x6 ≤ x7
    mv   x21, x10       // copy parameter x10 into x21
    mv   x22, x11       // copy parameter x11 into x22
    mv   x10, x21       // first swap parameter is v
    mv   x11, x20       // second swap parameter is j
    jal  x1,swap        // call swap
    addi x20,x20,-1     // j –= 1
    j    for2tst        // branch to test of inner loop
exit2:
```

Revised Code of The Inner Loop

- Skeleton of inner loop:
  - for (j = i − 1; j >= 0 && v[j] > v[j + 1]; j − = 1) {

```
    addi x20,x19,-1    // j = i –1
for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,3      // reg x5 = j * 8
    add  x5,x10,x5     // reg x5 = v + (j * 8)
    ld   x6,0(x5)      // reg x6 = v[j]
    ld   x7,8(x5)      // reg x7 = v[j + 1]
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
    mv   x22, x11      // copy parameter x11 into x22
    mv   x11, x20      // second swap parameter is j (pass j with x11)
    jal  x1,swap       // call swap (base address of v[] is in x10)
    mv   x11, x22      // Restore x11 (n)
    addi x20,x20,-1    // j –= 1
    j    for2tst       // branch to test of inner loop
exit2:
```

x11可能還有其他地方會使用到所以呼叫完函式後要立刻復原

x10不會改變

Preserving Registers

- Preserve saved registers:

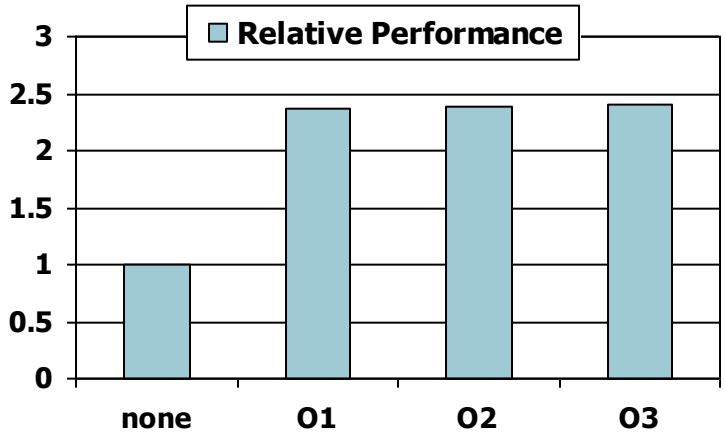
```
addi sp,sp,-40  // make room on stack for 5 regs
sd   x1,32(sp)  // save x1 on stack
sd   x22,24(sp) // save x22 on stack
sd   x21,16(sp) // save x21 on stack
sd   x20,8(sp)  // save x20 on stack
sd   x19,0(sp)  // save x19 on stack
```

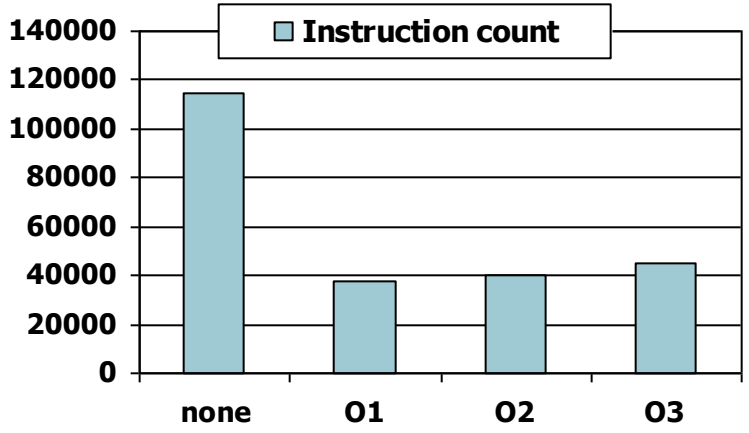
- Restore saved registers:

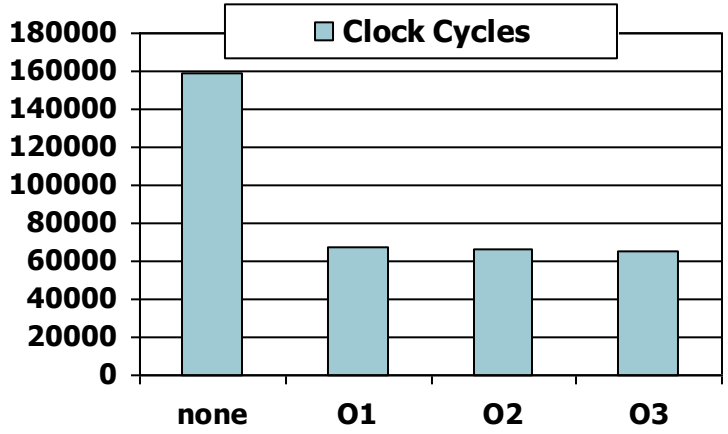
```
exit1:
  ld   x19,0(sp)  // restore x19 from stack
  ld   x20,8(sp)  // restore x20 from stack
  ld   x21,16(sp) // restore x21 from stack
  ld   x22,24(sp) // restore x22 from stack
  ld   x1,32(sp)  // restore x1 from stack
  addi sp,sp, 40  // restore stack pointer
  jalr x0,0(x1)
```

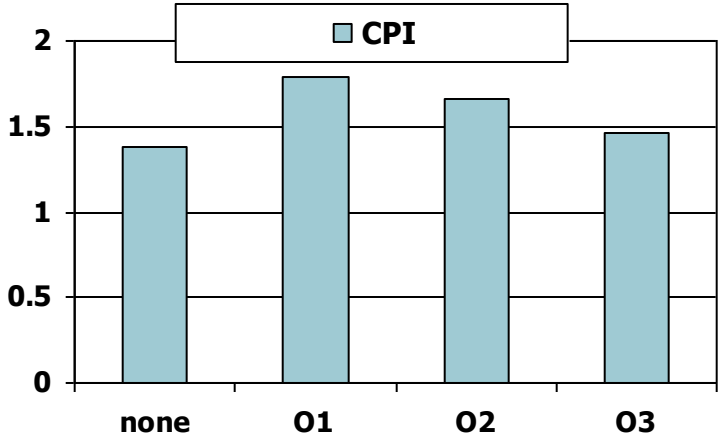
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

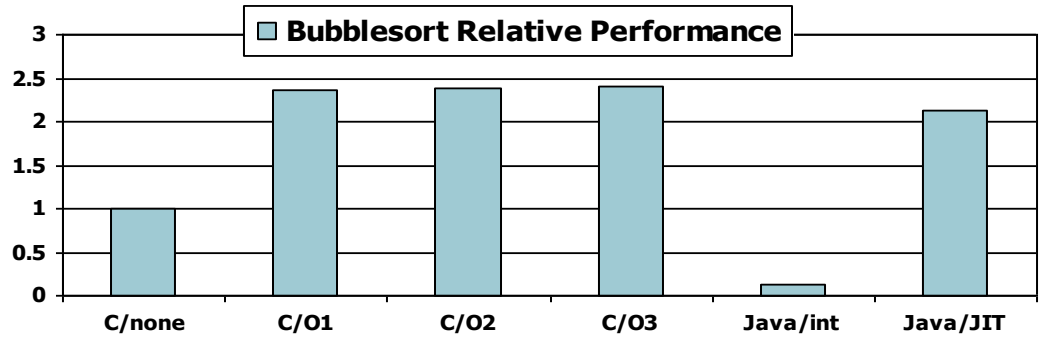


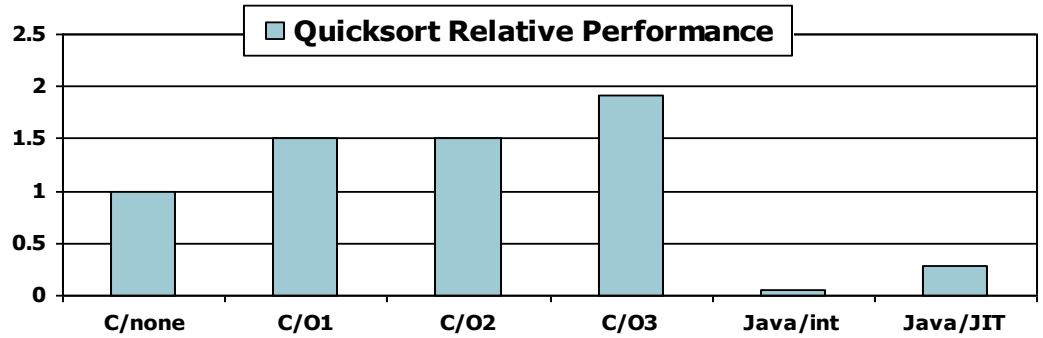


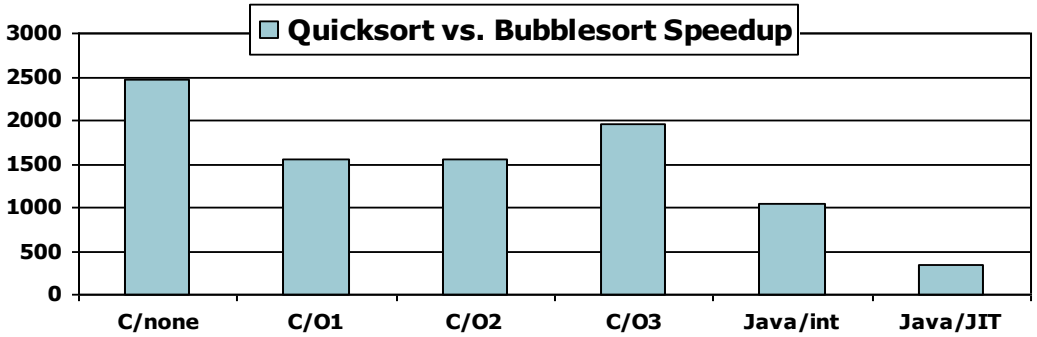




Effect of Language and Algorithm







Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing an Array

這邊比較快!

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
   li   x5,0        // i = 0
loop1:
   slli x6,x5,3     // x6 = i * 8
   add  x7,x10,x6   // x7 = address
                    // of array[i]
   sd   x0,0(x7)    // array[i] = 0
   addi x5,x5,1     // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

一格為Double Word (8 Bytes = 64bits)

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   mv x5,x10        // p = address
                    // of array[0]
   slli x6,x11,3    // x6 = size * 8
   add x7,x10,x6    // x7 = address
                    // of array[size]
loop2:
   sd x0,0(x5)      // Memory[p] = 0
   addi x5,x5,8     // p = p + 8
   bltu x5,x7,loop2
                    // if (p<&array[size])
                    // go to loop2
```

Array的範圍(Bytes個數)

若沒有超過Array的大小
(用記憶體位置比較)

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

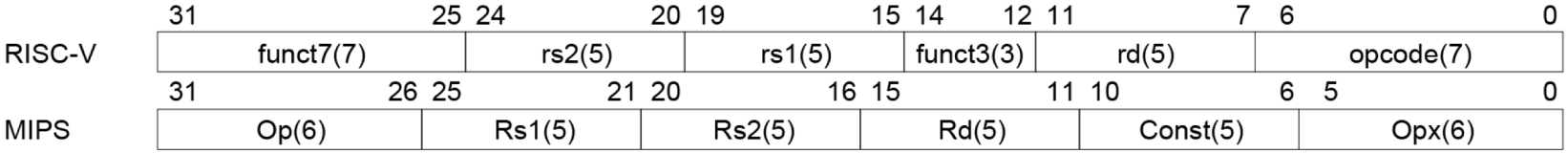
MIPS Instructions

- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
  - 32-bit instructions
  - 32 general purpose registers, register 0 is always 0
  - 32 floating-point registers
  - Memory accessed only by load/store instructions
    - Consistent use of addressing modes for all data sizes
- Different conditional branches
  - For <, <=, >, >=
  - RISC-V: blt, bge, bltu, bgeu
  - MIPS: slt, sltu (set less than, result is 0 or 1)
    - Then use beq, bne to complete the branch

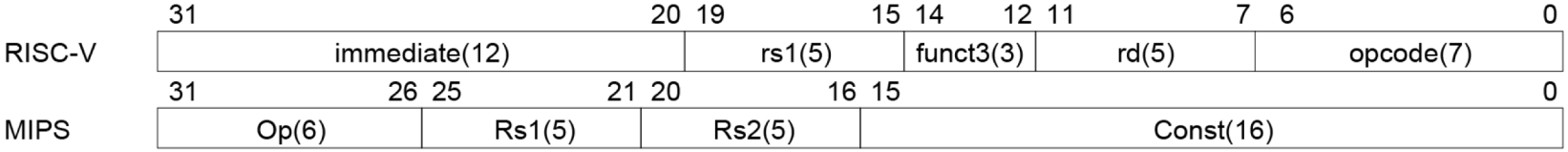
(1)RISC-V直接做比較並Branch
(2)MIPS則是先比較出結果，再用beq,bne Branch

Instruction Encoding

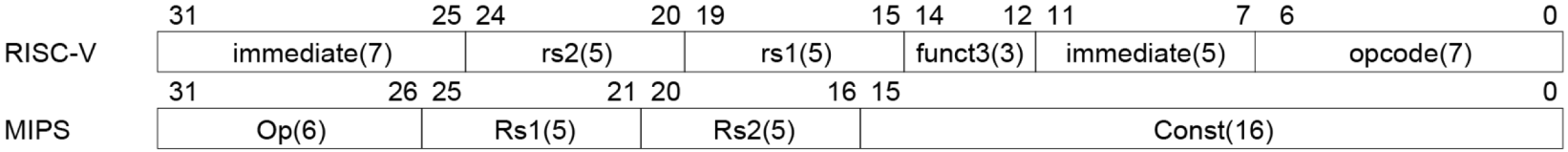
Register-register



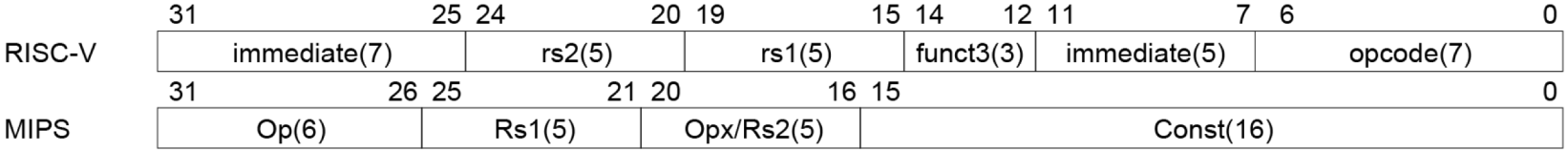
Load



Store



Branch



ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

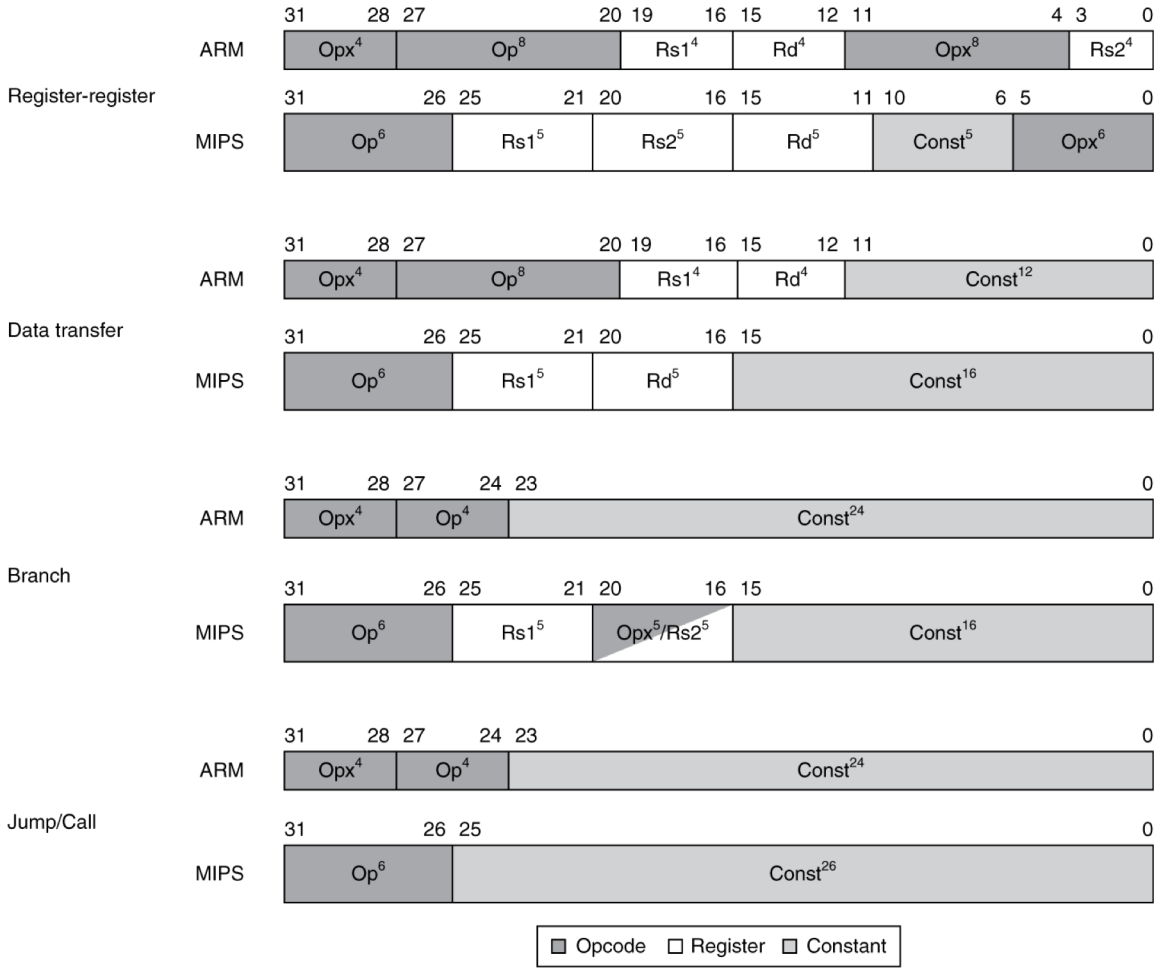
Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

```
Ex: If(true) { a =1; }
Assembly Code:
addi x5, x0, 1, "x6"
透過x6執行的結果去決定要不要
做，把If-else的Branch省去掉。
```

不浪費Branch在執行一個Instruction上面

Instruction Encoding



The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

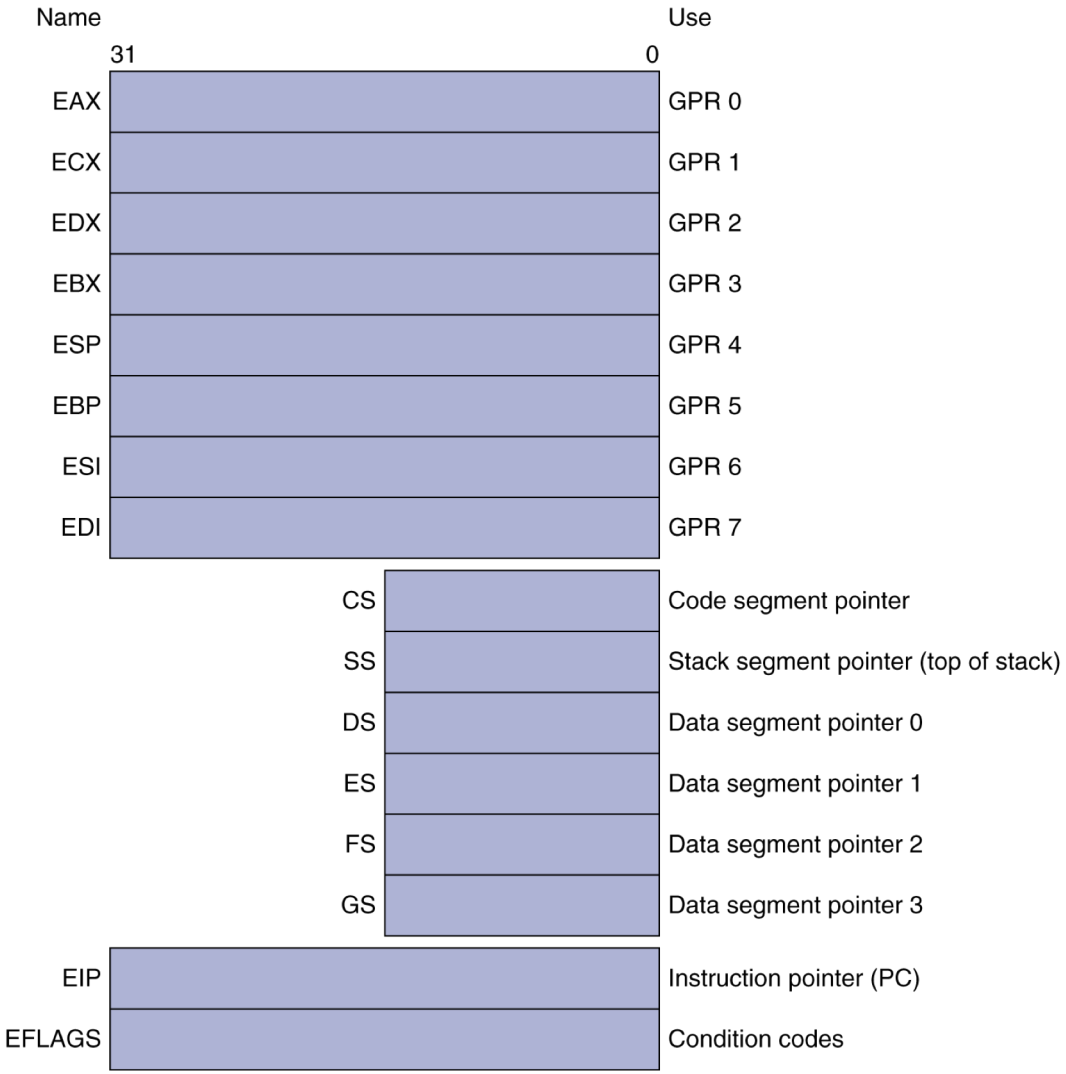
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



Basic x86 Addressing Modes

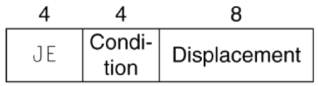
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

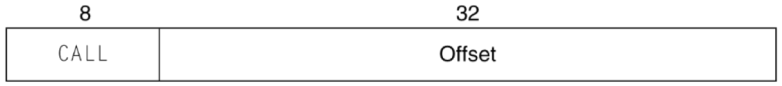
- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

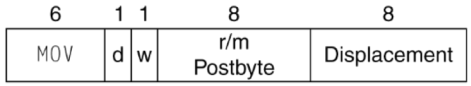
a. JE EIP + displacement



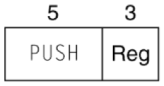
b. CALL



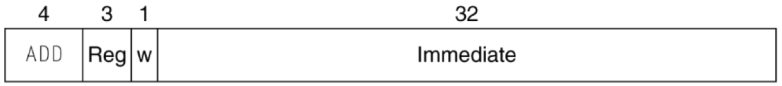
c. MOV EBX, [EDI + 45]



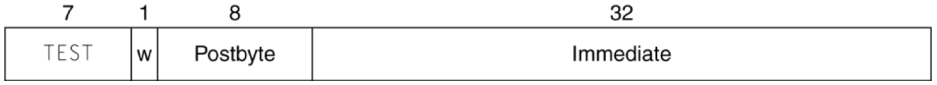
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

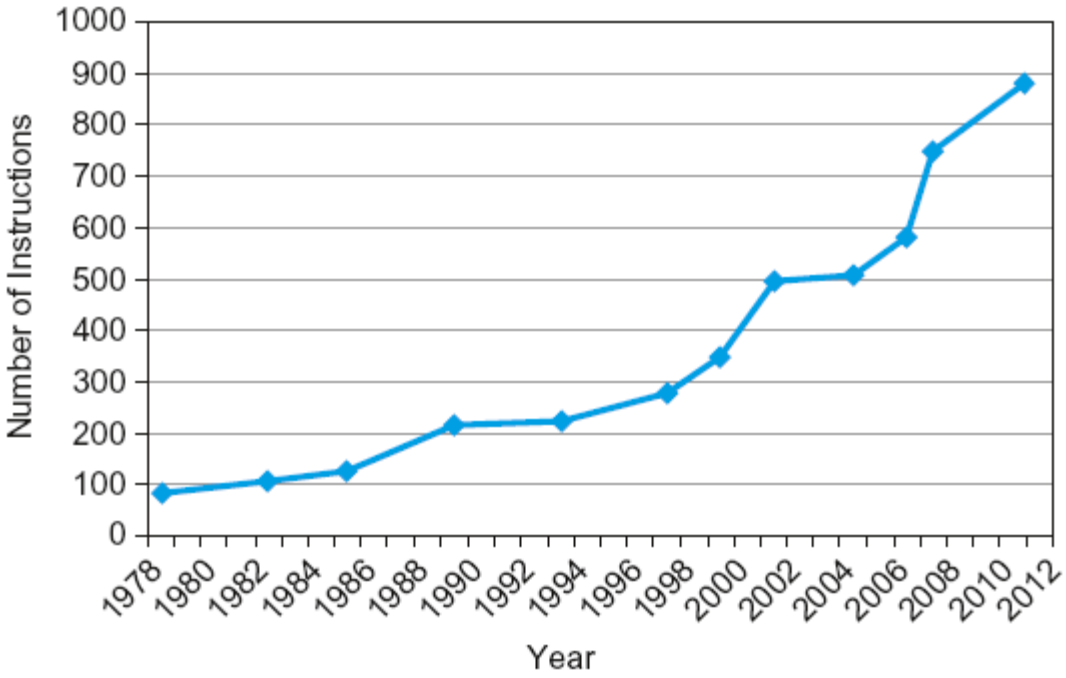
Fallacies

複雜Instruction設計難度高且會減慢其他Instruction，最好保持其simple並讓Compiler去做優化

- Powerful instruction ⇏ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
  1. Simplicity favors regularity 簡單有利於規律性
  2. Smaller is faster 越小越快
  3. Good design demands good compromises 綜合考慮
- Make the common case fast 常用的要速度快
- Layers of software/hardware
  - Compiler, assembler, hardware
- RISC-V: typical of RISC ISAs
  - c.f. x86