UNIVERSITÀ DEGLI STUDI DI TRIESTE

Dipartimento di Ingegneria e Architettura

Instructions: Language of the Computer

A. Carini – Digital System Architectures

Introduction

- To command a computer's hardware, you must speak its language.
- The words of a computer's language are called **instructions**, and its vocabulary is called an **instruction set**.
- Computer languages are quite similar, more like regional dialects than independent languages.
- The chosen instruction set is **ARMv8**, introduced in 2011. We will use a subset of ARMv8, called **LEGv8** ("Lessen Extrinsic Garrulity")**.**
- This similarity of instruction sets occurs because:
  - all computers are constructed based on similar underlying principles;
  - there are a few basic operations that all computers must provide;
  - computer designers have a common goal: the language should make it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy.

Operations of the Computer Hardware

- Every computer must be able to perform arithmetic.
- The LEGv8 notation for add the two variables *b* and *c* and to put their sum in *a*:

```
ADD a, b, c
```

- All arithmetic operations have this form.
- Suppose we want to place the sum of four variables *b, c, d*, and *e* into variable *a:*

```
ADD a, b, c       // The sum of b and c is placed in a
ADD a, a, d       // The sum of b, c, and d is now in a
ADD a, a, e       // The sum of b, c, d, and e is now in a
```

- Requiring every instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple:
  - hardware for a variable number of operands is more complicated than for a fixed number.
- ***Design Principle 1*****:** Simplicity favors regularity

Compiling a C Assignment into LEGv8

- C code: `f = (g + h) – (i + j);`
- What might a C compiler produce?

```
ADD t0,g,h // temporary variable t0 contains g + h
ADD t1,i,j // temporary variable t1 contains i + j
SUB f,t0,t1 // f gets t0 – t1, which is (g + h) – (i + j)
```

Operands of the Computer Hardware

- Arithmetic instructions use **register** operands.
- LEGv8 has a *32 × 64-bit register file*
  - Use for frequently accessed data
  - 64-bit data is called a "**doubleword**"
    - 32 x 64-bit general purpose registers **X0** to **X31**
  - 32-bit data called a "**word**"
    - 32 x 32-bit general purpose sub-registers **W0** to **W31**

- The reason for the limit of 32 registers may be found in

  - ***Design Principle 2*****:** Smaller is faster.

- A very large number of registers may increase the clock cycle time simply because it takes electronic signals longer when they must travel farther.
- The designer must balance the craving of programs for more registers with the designer's desire to keep the clock cycle fast.
- Another reason for not using more than 32 is the number of bits requested in the instruction format.

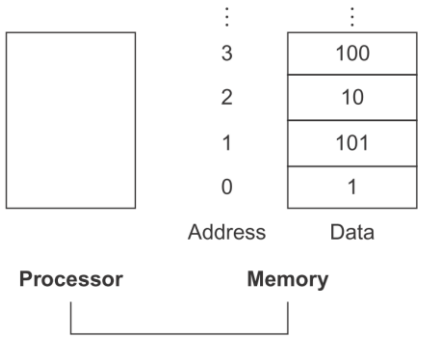
Compiling a C Assignment Using Registers

- C Code: `f = (g + h) – (i + j);`
- The variables *f, g, h, i,* and *j* are assigned to the registers X19, X20, X21, X22, and X23, respectively.
- What is the compiled LEGv8 code?

```
ADD X9,X20,X21  // register X9 contains g + h
ADD X10,X22,X23 // register X10 contains i + j
SUB X19,X9,X10  // f gets X9 – X10, which is (g + h) – (i + j)
```

Memory operands

- Programming languages have simple variables that contain single data elements, but they also have more complex data structures—arrays and structures.
- These composite data structures can contain many more data elements than there are registers.
- How can a computer represent and access such large structures?
- Arrays and structures are kept in memory.
- LEGv8 must include instructions that transfer data between memory and registers.
- Such instructions are called **data transfer instructions**.
- To access a word or doubleword in memory, the instruction must supply the memory address.
- *Memory* is just a *large, single-dimensional array,* with the address acting as the index to that array, starting at 0.



Memory operands

- The data transfer instruction that copies data from memory to a register is called **load**.
- The format of the load instruction is the name of the operation **LDUR**, **load register**, followed by the register to be loaded, then a **base register** and an **offset**, a constant used to access memory.
  - (U for unscaled)

- Let's assume that A is an array of 100 doublewords and that the compiler has associated the variables *g* and *h* with the registers X20 and X21. The base address of A is in X22.
- Compile this C statement:

```
g = h + A[8];
```

```
LDUR      X9,[X22,#8] // Temporary reg X9 gets A[8]
ADD       X20,X21,X9 // g = h + A[8]
```

Memory operands

- The data transfer instruction that copies data from memory to a register is called **load**.
- The format of the load instruction is the name of the operation **LDUR**, **load register**, followed by the register to be loaded, then a **base register** and an **offset**, a constant used to access memory.
  - (U for unscaled)

- Let's assume that A is an array of 100 doublewords and that the compiler has associated the variables *g* and *h* with the registers X20 and X21. The base address of A is in X22.
- Compile this C statement:

```
g = h + A[8];
```

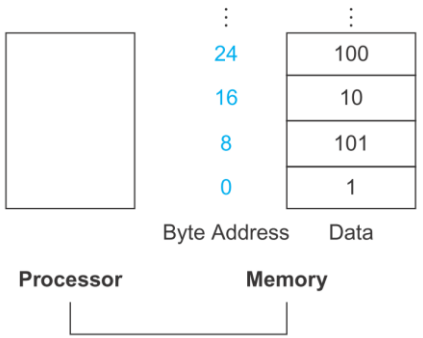
```
LDUR      X9,[X22,#8] // Temporary reg X9 gets A[8]
ADD       X20,X21,X9 // g = h + A[8]
```

There is an error…

Memory operands

- The compiler allocates data structures like arrays and structures to locations in memory.
- The compiler can then place the proper starting address into the data transfer instructions.
- Virtually all architectures today **address** individual **bytes.**
- The address of a doubleword matches the address of one of the 8 bytes within the doubleword, and addresses of sequential doublewords differ by 8.
- Thus,

```
LDUR      X9,[X22,#64] // 8 x 8 = 64
```



- Computers divide into those that use the address of the leftmost or “big end” byte as the doubleword address versus those that use the rightmost or “little end” byte.
- *LEGv8* can work either as *big-endian* or *little-endian*.
- *LEGv8* does not require words to be aligned in memory, except for instructions and the stack.

Memory operands

- The instruction complementary to load is traditionally called **store**; it copies data from a register to memory.
- The format of a store is similar to that of a load: the name of the operation, **STUR**, **store register**, followed by the register to be stored, the base register, and the offset to select the array element.

- Assume variable *h* is associated with register X21 and the base address of the array A is in X22.
- Compile

```
A[12] = h + A[8];
```

```
LDUR  X9, [X22,#64]  // Temporary reg X9 gets A[8]
ADD   X9,X21,X9      // Temporary reg X9 gets h + A[8]
STUR X9, [X22,#96] // Stores h + A[8] back into A[12]
```

Memory operands

- Many programs have more variables than computers have registers.
- *Registers are faster to access than memory*.
  - Operating on memory data requires loads and stores and more instructions to be executed.
- Compiler must use registers for variables as much as possible, spilling to memory for less frequently used variables.
- *Accessing registers also uses much less energy* than accessing memory.
- To achieve the highest performance and conserve energy, an instruction set architecture must have enough registers, and compilers must use registers efficiently.
  - *Register optimization is important!*

- Assuming 64-bit data, registers were roughly 200 times faster (0.25 ns vs. 50 ns) and 10,000 times more energy efficient (0.1 vs. 1000 picoJoules) than DRAM in 2015.
- These large differences led to caches, which reduce the performance and energy penalties of going to memory.

Constant or Immediate Operands

- Many times a program will use a constant in an operation.
- Using the instructions we have seen so far, we would have to load a constant from memory to use one.
- For example, to add the constant 4 to register X22:

```
LDUR X9, [X20, AddrConstant4]      // X9 = constant 4
ADD  X22,X22,X9                    // X22 = X22 + X9 (X9 == 4)
```

- An alternative is to offer versions of the arithmetic instructions in which one operand is a constant, as **ADDI**, **Add Immediate**

```
ADDI       X22,X22,#4              // X22 = X22 + 4
```

- Constant operands occur frequently, and by including constants inside arithmetic instructions, operations are much faster and use less energy than if constants were loaded from memory.

Clarifications

- Although the LEGv8 registers are 64 bits wide, the full ARMv8 instruction set has two execution states: **AArch32**, in which registers are 32 bits wide, and **AArch64**, which has a 64-bit wide register.
- The migration from 32-bit address computers to 64-bit address computers left compiler writers a choice of the size of data types in C. Clearly, pointers should be 64 bits, but what about integers?

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

  - We will use *long long int* for the 64bit words, *size_t* for indexes to arrays (it guarantees they are the right size no matter how big the array).
- In the full ARMv8 instruction set, register 31 is XZR in most instructions but the stack point (SP) in others. To avoid confusion, in LEGv8 **register 31 is always XZR** and **SP is always register 28**.
- The full ARMv8 instruction set does not use the mnemonic ADDI; it just uses **ADD**, and lets the assembler pick the proper opcode.

Representing Instructions in the Computer

- Instructions are encoded in binary code, called **machine code**
- We'll show the LEGv8 language version of the instruction represented symbolically as

```
ADD X9,X20,X21
```

- The decimal and the binary representation are

1112	21	0	20	9
10001011000	10101	000000	10100	01001
11 bits	5 bits	6 bits	5 bits	5 bits

$$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two} = 8B150289_{16}$$

- This layout of the instruction is called the **instruction format**. There are five **fields.**
- All LEGv8 instructions are **32 bits** long.

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

LEGv8 R-format Instructions

- **opcode:** operation code
- **Rm:** the second register source operand
- **shamt:** shift amount (00000 for now)
- **Rn:** the first register source operand
- **Rd:** the register destination

Good compromises

- A problem occurs when an instruction needs longer fields than those of the R format.
- For example, the *load register* instruction must specify two registers and a constant.
- If the address were to use one of the 5-bit fields in the format R, the largest constant within the load register instruction would be limited to only 2^{5-1} or 31.
- This constant is used to select elements from arrays or data structures, and it often needs to be much larger than 31.
- We have a conflict between the desire to keep all instructions the same length and the desire to have a single instruction format.

    - **Design Principle 3:** *Good design demands good compromises.*

- The compromise chosen by the LEGv8 designers is to keep all instructions the same length, thereby requiring distinct instruction formats for different kinds of instructions.
    - Different formats complicate decoding, but allow 32-bit instructions uniformly
    - Keep formats as similar as possible

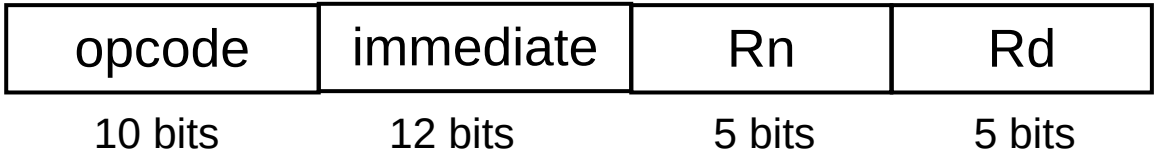
LEGv8 D-format Instructions

- Load/store instructions
  - Rn: base register
  - address: constant offset from contents of base register (±256 bytes, i.e., +/- 32 doublewords)
  - Rt: destination (load) or source (store) register number

```
LDUR X9, [X22,#64] // Temporary reg X9 gets A[8]
```

  - Opcode = 1986, Rn = 22, address = 64, Rt = 9

LEGv8 I-format Instructions



- Immediate instructions
  - Rn: source register
  - Rd: destination register
- Immediate field is zero-extended. Thus, only positive immediates!

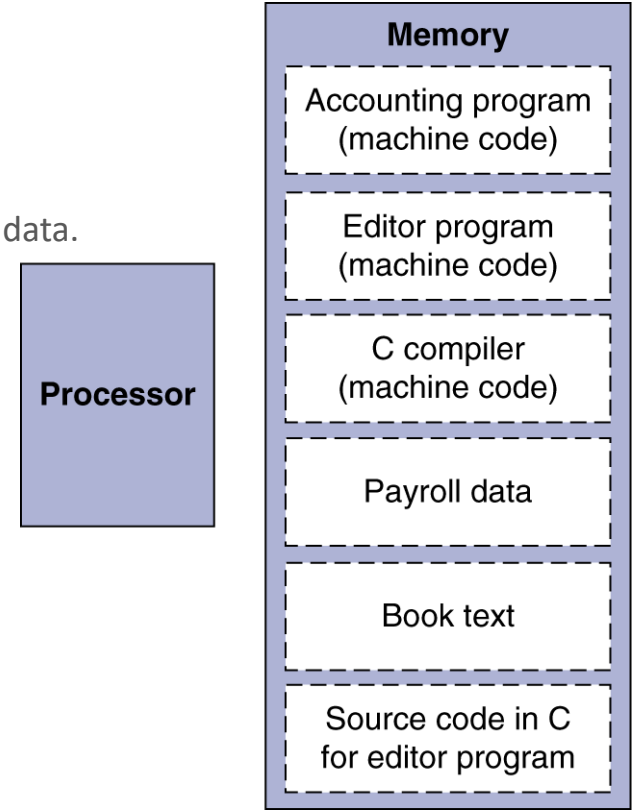
LEGv8 Instructions seen so far

LEGv8

Name	Format	Example						Comments
ADD	R	1112	3	0		2	1	ADD X1, X2, X3
SUB	R	1624	3	0		2	1	SUB X1, X2, X3
ADDI	I	580	100			2	1	ADDI X1, X2, #100
SUBI	I	836	100			2	1	SUBI X1, X2, #100
LDUR	D	1986	100		0	2	1	LDUR X1, [X2, #100]
STUR	D	1984	100		0	2	1	STUR X1, [X2, #100]
Field size		11 or 10 bits	5 bits	5 or 4 bits	2 bits	5 bits	5 bits	All ARM instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format

Stored Program Computers

- Today's computers are built on two key principles:
  - 1. Instructions are represented as numbers.
  - 2. Programs are stored in memory to be read or written, just like data.
- These principles lead to the **stored-program** concept;
  - Memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code. Programs can operate on programs.
- Programs are often shipped as files of binary numbers.
- Computers can inherit ready-made software provided they are compatible with an existing instruction set.
- Such "**binary compatibility**" often leads industry to align around a small number of instruction set architectures.



Logical operations

- Highly used for packing and unpacking of bits into words.

Logical operations	C operators	Java operators	LEGv8 instructions
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND,ANDI
Bit-by-bit OR	|	|	ORR,ORRI
Bit-by-bit NOT	~	~	EOR,EORI

FIGURE 2.8 C and Java logical operators and their corresponding LEGv8 instructions. One way to implement NOT is to use EOR with one operand being all ones (FFFF FFFF FFFF $FFFF_{hex}$).

Logical operations

- The first class of such operations is called shifts.
- They move all the bits in a doubleword to the left (*logical shift left* LSL) or right (logical shift right LSR), filling the emptied bits with 0s.
- Example

X19 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{two} = 9_{ten}$

```
LSL X11,X19,#4 // reg X11 = reg X19 << 4 bits
```

X11 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 10010000_{two} = 144_{ten}$

opcode	Rm	shamt	Rn	Rd
1691	0	4	19	11

Logical operations

- Another useful operation that isolates fields is AND.
- AND is a bit-by-bit operation that leaves a 1 in the result only if both bits of the operands are 1.
- Example:

```
        AND X9,X10,X11      // reg X9 = reg X10 & reg X11
```

X11 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101\ 11000000_{two}$

X10 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111100\ 00000000_{two}$

X9 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001100\ 00000000_{two}$

- OR is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1.

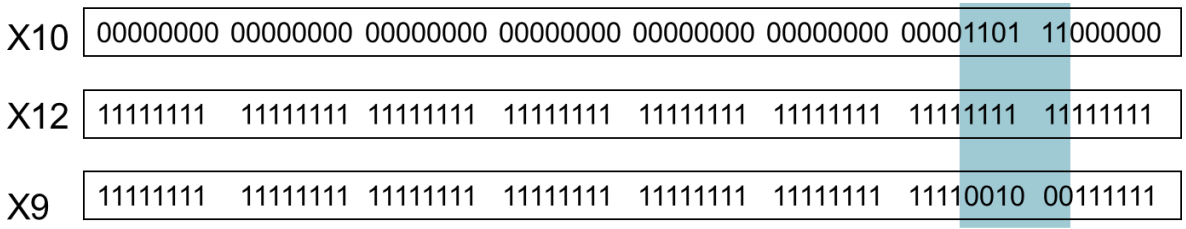
```
        ORR X9,X10,X11 // reg X9 = reg X10 | reg X11
```

X9 $00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111101\ 11000000_{two}$

Logical operations

- NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa.
- In keeping with the three-operand format, the designers of ARMv8 decided to include the instruction **EOR** (Exclusive OR) instead of NOT.
- Since exclusive OR creates a 0 when bits are the same and a 1 if they are different, the equivalent to NOT is an **EOR 111…111**.

```
EOR X9,X10,X12  // NOT operation
```



- Constants are useful in logical operations as well as in arithmetic operations, so LEGv8 also provides the instructions and immediate (**ANDI**), or immediate (**ORRI**), and exclusive or immediate (**EORI**).

Differences between ARMv8 and LEGv8

- The immediate fields for ANDI, ORRI, and EORI of the full ARMv8 instruction set are not simple 12-bit immediates.
- ARMv8 has the unusual feature of using a complex algorithm for encoding immediate values.
- Some small constants (e.g., 1, 2, 3, 4, and 6) are valid, while others (e.g., 5) are not.
- LEGv8 simply uses normal 12-bit immediates as found in ADDI.
- This difference means **EORI X1,X1,#5** is **legal for LEGv8** but **not ARMv8**.

- Unlike almost all other computer architectures, ARMv8 allows a register to be shifted as part of an arithmetic or logical instruction.
- Since this combination is unusual in computer architectures and not frequently generated by compilers, LEGv8 treat shifts as separate instructions.
- The opcode used is that of UBFM (unsigned bitfield move) but the Rm and shamt fields coding has been simplified.

Instructions for Making Decisions

- Decision making is commonly represented in programming languages using the *if* statement, sometimes combined with *go to* statements and labels.
- LEGv8 includes two decision-making instructions, similar to an *if* statement with *a go to*.

```
CBZ register, L1
```

  - means go to the statement labeled *L1* if the value in register equals zero.
  - CBZ stands for *compare and branch if zero*.

```
CBNZ register, L1
```

  - means go to the statement labeled *L1* if the value in register does not equal zero.
  - CBNZ stands for *compare and branch if not zero*.

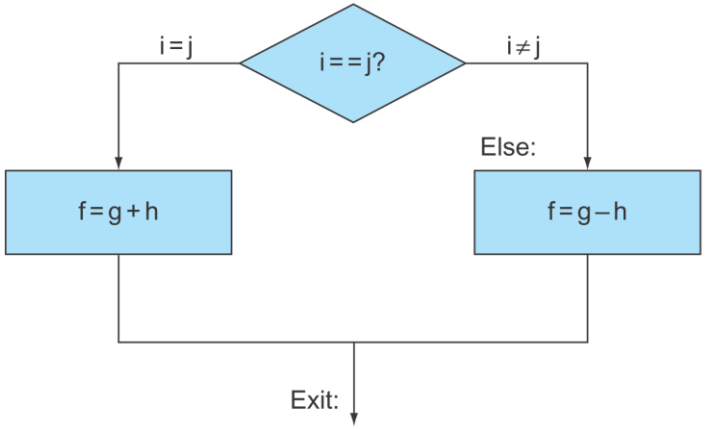
- These two instructions are traditionally called **conditional branches.**

Compiling *if-then-else* into Conditional Branches

- *f, g, h, i,* and *j* are variables that correspond to the five registers X19 through X23.
- What is the compiled LEGv8 code for this C if statement?

```
if (i == j) f = g + h; else f = g – h;
```

```
    SUB  X9,X22,X23 // X9 = i – j
    CBNZ X9, Else   // go to Else if i ≠ j (X9 ≠ 0)
    ADD X19,X20,X21       // f = g + h (skipped if i ≠ j)
    B Exit        // go to Exit
Else:SUB X19,X20,X21    // f = g – h (skipped if i = j)
Exit:
```



Assembler calculates addresses

Compiling a *while* Loop in C

- Here is a traditional loop in C:

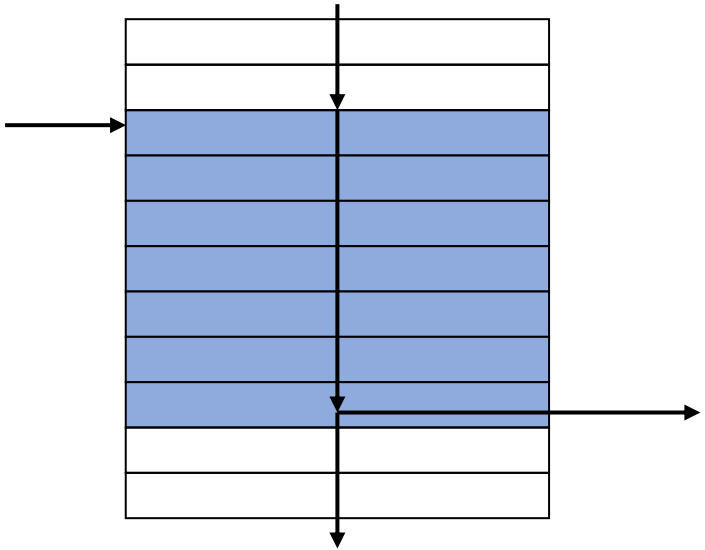
```
while (save[i] == k)
    i += 1;
```

- Assume that i and k correspond to registers X22 and X24 and the base of the array save is in X25.
- What is the LEGv8 assembly code corresponding to this C code?

```
Loop: LSL X10,X22,#3      // Temp reg X10 = i * 8
      ADD X10,X10,X25     // X10 = address of save[i]
      LDUR X9, [X10,#0]   // Temp reg X9 = save[i]
      SUB X11,X9,X24      // X11 = save[i] – k
      CBNZ X11, Exit      // go to Exit if save[i] ≠ k(X11 ≠ 0)
      ADDI X22,X22,#1     // i = i + 1
      B       Loop        // go to Loop
Exit:
```

Basic Blocks

- **basic block** A sequence of instructions with
  - No branches (except possibly at the end) and
  - No branch targets or branch labels (except possibly at the beginning).



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- The full set of comparisons is less than (<), less than or equal (≤), greater than (>), greater than or equal (≥), equal (=), and not equal (≠).
- Comparisons must also deal with the dichotomy between *signed* and *unsigned numbers*.
- **Condition codes** or **flags** are used to handle all these cases:
  - **negative (N)** – the result that set the condition code had a 1 in the most significant bit;
  - **zero (Z)** – the result that set the condition code was 0;
  - **overflow (V)** – the result that set the condition code overflowed; and
  - **carry (C)** – the result that set the condition code had a carry out of the most significant bit or a borrow into the most significant bit.
- They are set by a limited number of operations —ADD, ADDI, AND, ANDI, SUB, and SUBI— when the condition code is activated.
- In LEGv8 assembly language, append an S to the end of one of these instructions if you want to set condition codes: **ADDS, ADDIS, ANDS, ANDIS, SUBS,** and **SUBIS.**

More Conditional Operations

- Conditional branches (written as **B.cond**) use combinations of the condition codes.
- Use subtract to set flags, then conditionally branch:
  - B.EQ
  - B.NE
  - B.LT (less than, signed), B.LO (less than, unsigned)
  - B.LE (less than or equal, signed), B.LS (less than or same, unsigned)
  - B.GT (greater than, signed), B.HI (greater than, unsigned)
  - B.GE (greater than or equal, signed), B.HS (greater than or same, unsigned)

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

Conditional Example

- C code:
  ```
  if (a > b) a += 1;
  ```
  a in X22, b in X23

- The corresponding LEGv8 assembly code is:

```
      SUBS X9,X22,X23  // use subtract to make comparison
      B.LE Exit                // conditional branch
      ADDI X22,X22,#1
Exit: …
```

Supporting Procedures in Computer Hardware

- **procedure** A stored subroutine that performs a specific task based on the parameters with which it is provided.
- Procedures are one way to implement *abstraction* in software.
- In the execution of a procedure, the program must follow these six steps:
    1. Put parameters in a place where the procedure can access them.
    2. Transfer control to the procedure.
    3. Acquire the storage resources needed for the procedure.
    4. Perform the desired task.
    5. Put the result value in a place where the calling program can access it.
    6. Return control to the point of origin, since a procedure can be called from several points in a program.

LEGv8 Support for Procedures

- LEGv8 software follows the following convention for procedure calling in allocating its 32 registers:
  - X0–X7: eight parameter registers in which to pass parameters or return values.
  - **LR** (X30): one return address register to return to the point of origin.
- LEGv8 assembly language includes an instruction just for the procedures:
  - **branch-and-link instruction (BL)**

```
BL ProcedureAddress
```

  - it branches to an address and simultaneously saves the address of the following instruction, i.e., the **return address** in register LR (X30).
- To support the return from a procedure, computers like LEGv8 use the branch register instruction (BR) meaning an unconditional branch to the address specified in a register:

```
BR    LR
```

- Implicit in the stored-program idea is the need to have a register to hold the address of the current instruction being executed, the **program counter.**

Using more registers

- Suppose a compiler needs more registers for a procedure than the eight argument registers.
- Any registers needed by the caller must be restored to the values that they contained before the procedure was invoked.
- The ideal data structure for spilling registers is a **stack**—a last-in-first-out queue.
- A stack needs a pointer, the **stack pointer**, to the most recently allocated address in the stack.
- The **stack pointer (SP),** which is just one of the 32 registers, is adjusted by one doubleword for each register that is saved or restored.
    - SP is X28 in LEGv8, but is X31 in ARMv8.
- Placing data onto the stack is called a **push**, and removing data from the stack is called a **pop**.
    - By historical precedent, stacks “grow” from higher addresses to lower addresses.
    - This convention means that you push values onto the stack by subtracting from the stack pointer.
    - Adding to the stack pointer shrinks the stack, thereby popping values off the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

```
long long int leaf_example (long long int g, long long
int h, long long int i, long long int j)
{
        long long int f;

        f = (g + h) – (i + j);
        return f;
}
```

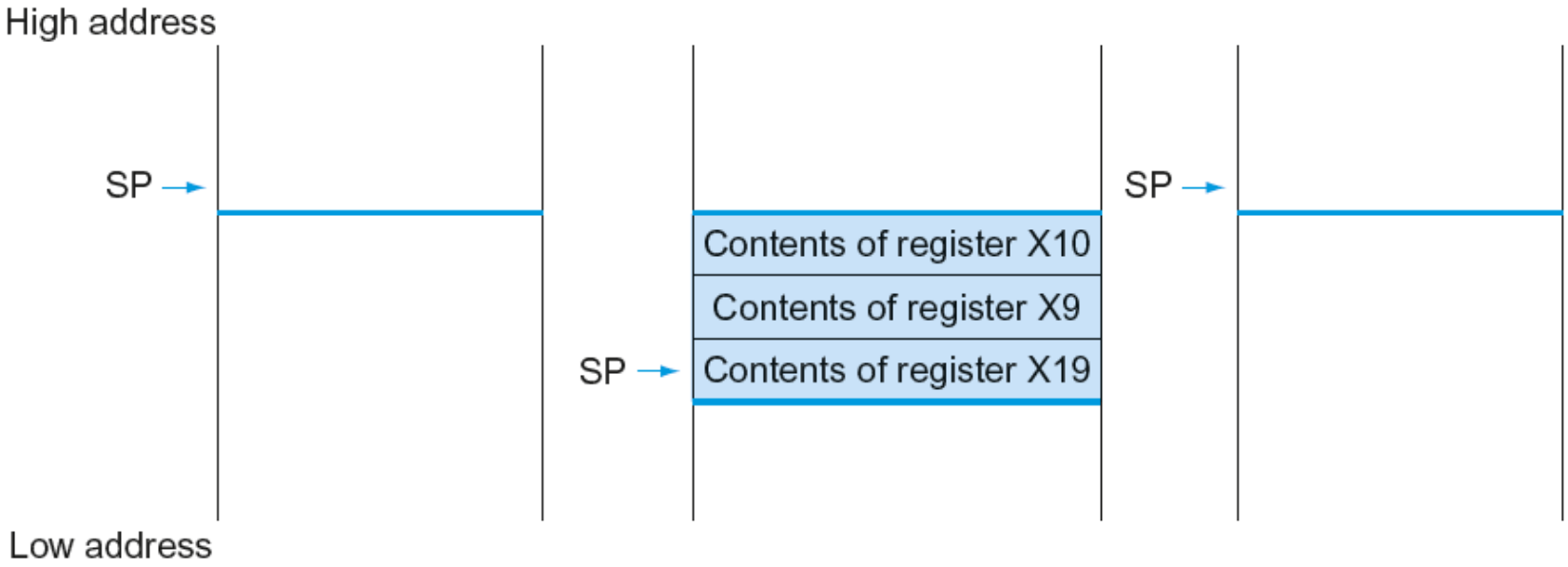
- The parameter variables g, h, i, and j correspond to the argument registers X0, X1, X2, and X3,
- f corresponds to X19.

- We will use X9, X10, X19: they could be needed by the caller. Thus, we will "push" them into the stack.

Compiling a C Procedure That Doesn't Call Another Procedure

```
leaf_example:
     SUBI SP,  SP, #24        // adjust stack to make room for 3 items
     STUR X10, [SP,#16]       // save register X10 for use afterwards
     STUR X9,  [SP,#8]        // save register X9 for use afterwards
     STUR X19, [SP,#0]        // save register X19 for use afterwards

     ADD X9,X0,X1   // register X9 contains g + h
     ADD X10,X2,X3  // register X10 contains i + j
     SUB X19,X9,X10 // f = X9 - X10, which is (g + h) - (i + j)

     ADD X0,X19,XZR // returns f (X0 = X19 + 0)

     LDUR X19, [SP,#0]     // restore register X19 for caller
     LDUR X9, [SP,#8]      // restore register X9 for caller
     LDUR X10, [SP,#16]   // restore register X10 for caller
     ADDI SP,SP,#24       // adjust stack to delete 3 items

     BR      LR           // branch back to calling routine
```

Local Data on the Stack



Register Usage

- We have used temporary registers and assumed their old values must be saved and restored.
- To avoid saving and restoring a register whose value is never used, LEGv8 software separates 19 of the registers into two groups:
  - X9–X18: temporary registers that are not preserved by the callee (called procedure) on a procedure call;
  - X19–X27: saved registers that must be preserved on a procedure call (if used, the callee saves and restores them).

Non-Leaf Procedures

- Non-Leaf Procedures are procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address LR
  - Any arguments (X0-X7) and temporaries (X9-X17) needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

- Argument **n** in X0
- ~~Result in X1 (<<< unusual !!! Should be in X0)~~
- Result in X0

Non-Leaf Procedure Example

```
fact:
     SUBI  SP, SP, #16 // adjust stack for 2 items
     STUR  LR, [SP,#8] // save the return address
     STUR  X0, [SP,#0] // save the argument n

     SUBIS     XZR,X0, #1  // test for n < 1
     B.GE      L1          // if n >= 1, go to L1

     ADDI    X0,XZR, #1  // return 1
     ADDI    SP,SP,#16  // pop 2 items off stack
     BR      LR       // return to caller

L1:  SUBI X0,X0,#1  // n >= 1: argument gets (n - 1)
     BL fact       // call fact with (n - 1)

     LDUR    X1, [SP,#0]    // return from BL: restore argument n
     LDUR    LR, [SP,#8]    // restore the return address
     ADDI    SP, SP, #16    // adjust stack pointer to pop 2 items
     MUL     X0,X0,X1       // return n * fact (n - 1)
     BR      LR       // return to the caller
```

Preserved/Not Preserved between calls

Preserved	Not preserved
Saved registers: X19-X27	Temporary registers: X9-X15
Stack pointer register: X28(SP)	Argument/Result registers: X0-X7
Frame pointer register: X29(FP)	
Link Register (return address): X30(LR)	
Stack above the stack pointer	Stack below the stack pointer

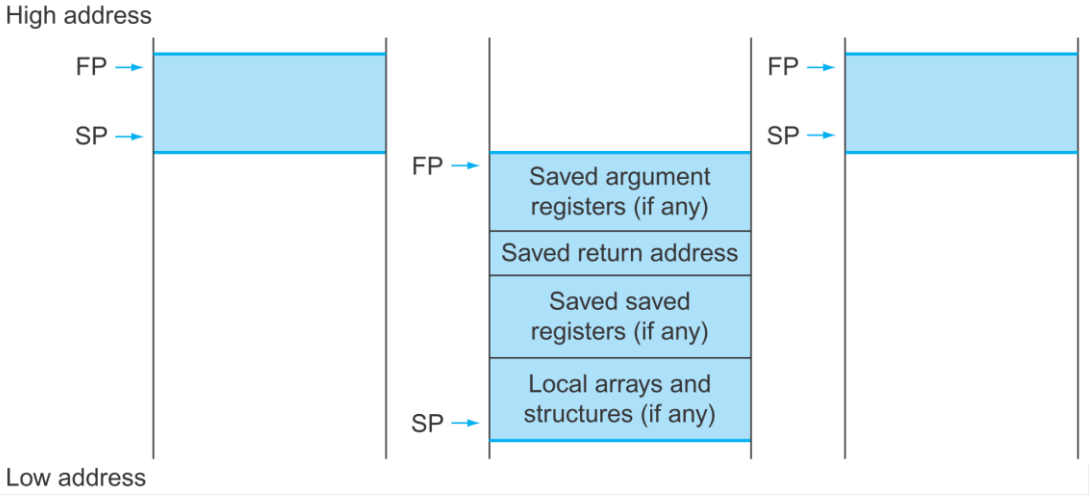
Managing C variables

- A C variable is generally a location in storage, and its interpretation depends both on its *type* and *storage class*.
- Example types include integers and characters.
- C has two storage classes: *automatic* and *static*.
- Automatic variables are local to a procedure and are discarded when the procedure exits.
- Static variables exist across exits from and entries to procedures.
    - declared outside all procedures or using the keyword *static*
- To simplify access to static data, some LEGv8 compilers reserve a register, called the **global pointer**, or GP, e.g., X27.

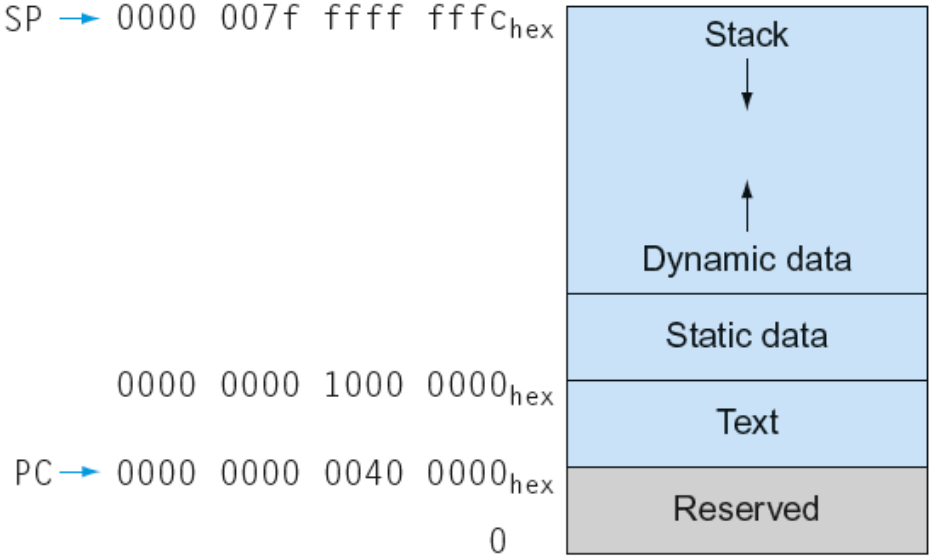
- Automatic variables are saved in registers or stack.
- Static variables are saved in the static data segment.
- Dynamically allocated memory is placed in the **heap.**

Procedure frame

- The segment of the stack containing a procedure's saved registers and local variables is called a **procedure frame** or **activation record**.
- Some ARMv8 compilers use a **frame pointer (FP)** to point to the first doubleword of the frame of a procedure.
- A stack pointer might change during the procedure.
- A frame pointer offers a stable base register within a procedure for local memory-references.



Memory Layout



- **Text**: program code
- **Static data**: global variables
  - e.g., static variables in C, constant arrays and strings
- **Dynamic data:** heap
  - E.g., malloc in C, new in Java
- **Stack**: automatic storage

Summary of Register Conventions

Name	Register number	Usage	Preserved on call?
X0-X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9-X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19-X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

Byte/Halfword Operations

- LEGv8 byte/halfword load/store
  - Load byte:
    - `LDURB Rt, [Rn, offset]`
    - ~~Sign extend to 64 bits in Rt (???)~~
  - Store byte:
    - `STURB Rt, [Rn, offset]`
    - Store just rightmost byte
  - Load halfword:
    - `LDURH Rt, [Rn, offset]`
    - ~~Sign extend to 64 bits in Rt (???)~~
  - Store halfword:
    - `STURH Rt, [Rn, offset]`
    - Store just rightmost halfword
- LEGv8 word load/store
  - Load signed word (signed extended to 64 bits):
    - `LDURSW Rt, [Rn, offset]`
  - Store word:
    - `STURW Rt, [Rn, offset]`

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;
}
```

  - Base addresses for arrays x and y are found in X0 and X1, while i is in X19.

String Copy Example

- LEGv8 code:

```
strcpy:
    SUBI SP,SP,8              // push X19
    STUR X19,[SP,#0]
    ADD X19,XZR,XZR           // i=0
L1: ADD X10,X19,X1            // X10 = addr of y[i]
    LDURB X11,[X10,#0]        // X11 = y[i]
    ADD X12,X19,X0            // X12 = addr of x[i]
    STURB X11,[X12,#0]        // x[i] = y[i]
    CBZ X11,L2                // if y[i] == 0 then exit
    ADDI X19,X19,#1           // i = i + 1
    B L1                      // next iteration of loop
L2: LDUR X19,[SP,#0]          // restore saved $s0
    ADDI SP,SP,8              // pop 1 item from stack
    BR LR                     // and return
```

Note that…

- ARMv8 software is required to keep the stack aligned to “quadword” (16 byte) addresses to get better performance. This convention means that a single char variable allocated on the stack occupies 16 bytes, even though it needs less. However, a C string variable or an array of bytes will pack 16 bytes per quadword.

- LEGv8 keeps everything 64 bits vs. providing both 32-bit and 64-bit address instructions as in ARMv8, which means it needs to include STURW (store word) as an instruction even though it is not specified in ARMv8 in assembly language. ARMv8 just uses STUR with a W register name (32-bit register) instead of X register name (64-bit register).

Wide Immediate Operands

- Most constants are small and the 12-bit immediate is sufficient.
- For the occasional 32-bit constant
- **MOVZ**: move wide with zeros
- **MOVK**: move wide with keep
- can set any 16 bits of a constant in a register.
- The 16-bit field to be loaded is specified by adding LSL and then the number 0, 16, 32, or 48.

The machine language version of `MOVZ X9, 255, LSL 16`:

110100101	01	0000 0000 1111 1111	01001

IW format

Contents of register `X9` after executing `MOVZ X9, 255, LSL 16`:

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000

The machine language version of `MOVK X9, 255, LSL 0`:

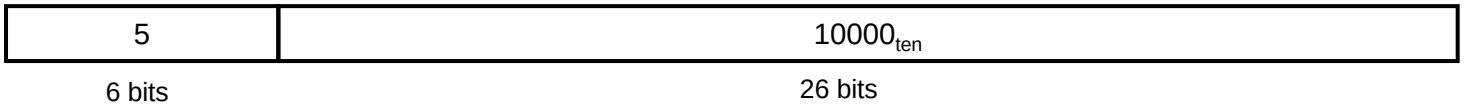
111100101	00	0000 0000 1111 1111	01001

Given value of `X9` above, new contents of `X9` after executing `MOVK X9, 255, LSL 0`:

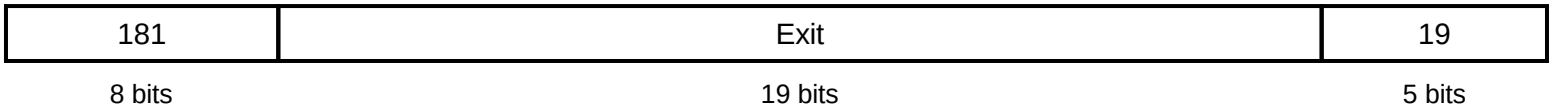
0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 1111 1111

Addressing in Branches

- B-type
  - `B 1000 // go to location 10000`$_{ten}$



- CB-type
  - `CBNZ X19, Exit // go to Exit if X19 != 0`



- Both addresses are PC-relative
  - Address = PC + offset (from instruction)
- This form of branch addressing is called **PC-relative addressing**.
- Since all LEGv8 instructions are 4 bytes long, LEGv8 stretches the branch distance by having PC-relative addressing refer to the **number of words** to the next instruction instead of the number of bytes.
  - the 19-bit field can branch ±1 MB from the current PC
  - the 26-bit field can branch ±128 MB from the current PC

Addressing in Branches

- Most conditional branches are to a nearby location, but occasionally they branch far away, farther than can be represented in the 19 bits of the conditional branch instruction.
- The assembler comes to the rescue!
- Given

```
        CBZ    X19, L1
```

- It can replace the short-address conditional branch with

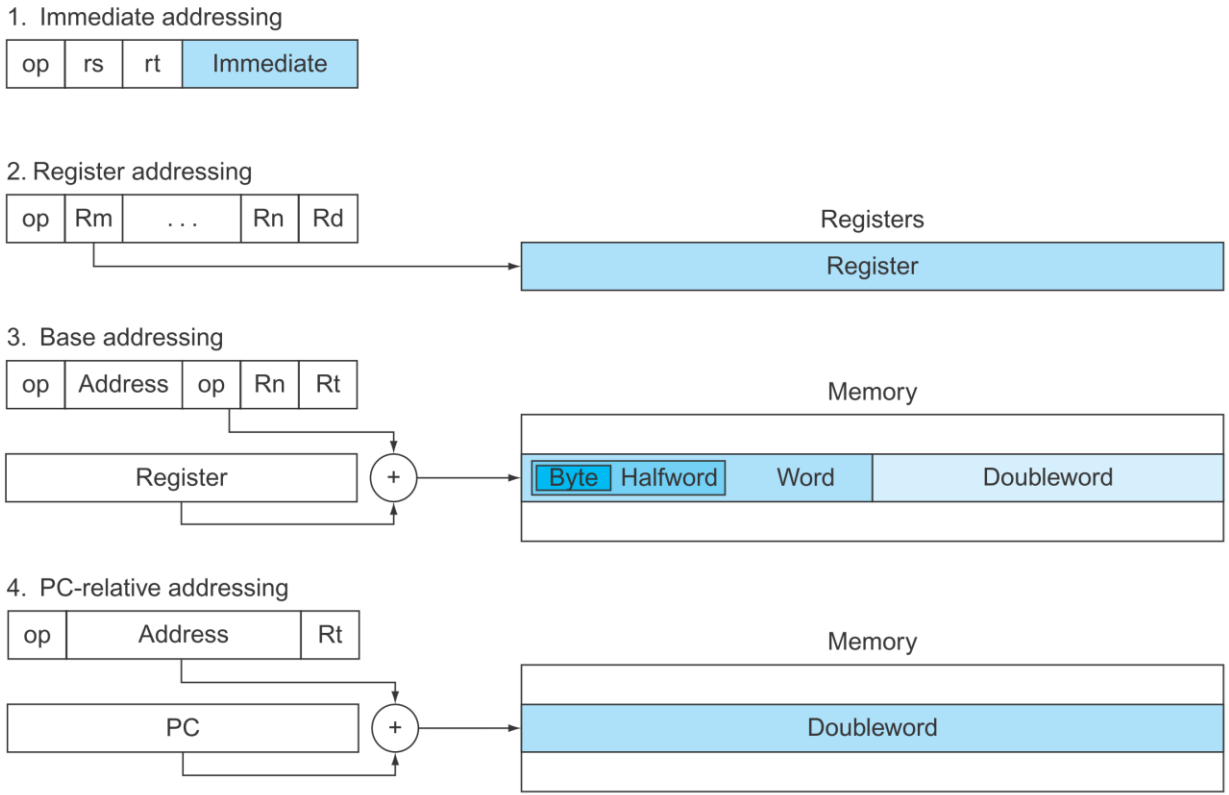
```
        CBNZ   X19, L2
        B      L1
L2:
```

LEGv8 Addressing Mode Summary

- The addressing modes of the LEGv8 instructions are the following:

    1. *Immediate addressing*, where the operand is a constant within the instruction itself.
    2. *Register addressing*, where the operand is a register.
    3. *Base or displacement addressing*, where the operand is at the memory location whose address is the sum of a register and a constant in the instruction, e.g. LDUR, STUR
    4. *PC-relative addressing*, where the branch address is the sum of the PC and a constant in the instruction.

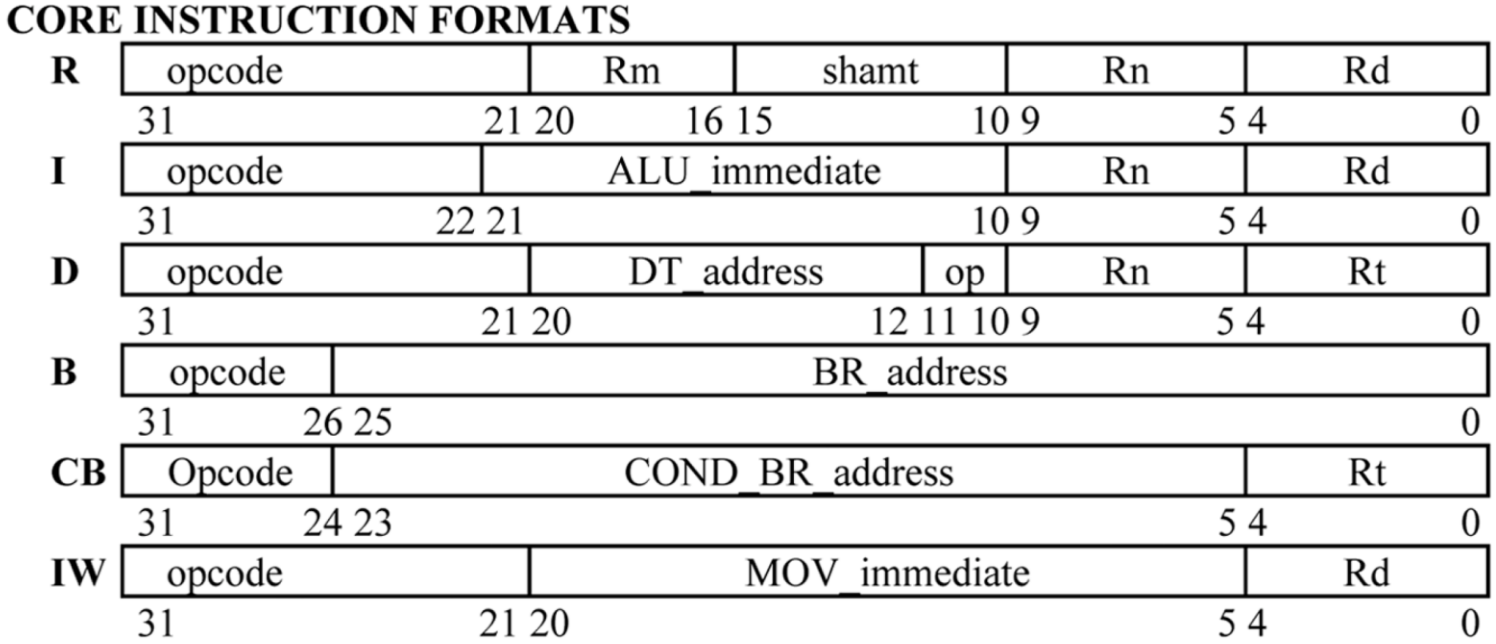
LEGv8 Addressing Mode Summary



LEGv8 Encoding Summary

Name	Fields							Comments
Field size		6 to 11 bits	5 to 10 bits	5 or 4 bits	2 bits	5 bits	5 bits	All LEGv8 instructions are 32 bits long
R-format	R	opcode	Rm	shamt		Rn	Rd	Arithmetic instruction format
I-format	I	opcode	immediate			Rn	Rd	Immediate format
D-format	D	opcode	address		op2	Rn	Rt	Data transfer format
B-format	B	opcode	address					Unconditional Branch format
CB-format	CB	opcode	address				Rt	Conditional Branch format
IW-format	IW	opcode	immediate				Rd	Wide Immediate format

LEGv8 Encoding Summary



Parallelism and Instructions: Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends on 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

- Assume that we want to build a simple lock where the value 0 is used to indicate that the lock is free and 1 is used to indicate that the lock is unavailable.
- A processor tries to set the lock by doing an exchange of 1, which is in a register, with the memory address corresponding to the lock.
- The value returned from the exchange instruction is 1 if some other processor had already claimed access, and 0 otherwise.
- In the latter case, the value is also changed to 1, preventing any competing exchange in another processor from also retrieving a 0.

Synchronization in LEGv8

- LEGv8 includes a special load and a special store called:
  - load exclusive register (**LDXR**)
  - store exclusive register (**STXR**)
- These instructions are used in sequence.
- If the contents of the memory location specified by LDXR are changed before the STXR to the same address occurs, then the STXR fails and does not write the value to memory.
- The STXR is defined to both store the value of a register in memory and to change the value of another register to a 0 if it succeeds and to a 1 if it fails.
- STXR specifies three registers: one to indicate failure or success, one to hold the value to be stored in memory, and one to hold the address.

Synchronization in LEGv8

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

```
again:LDXR X10,[X20,#0]      // load exclusive
      STXR X23, X9, [X20]    // store exclusive
      CBNZ X9,again          // branch if store fails
      ADD  X23,XZR,X10       // put loaded value in X23
```

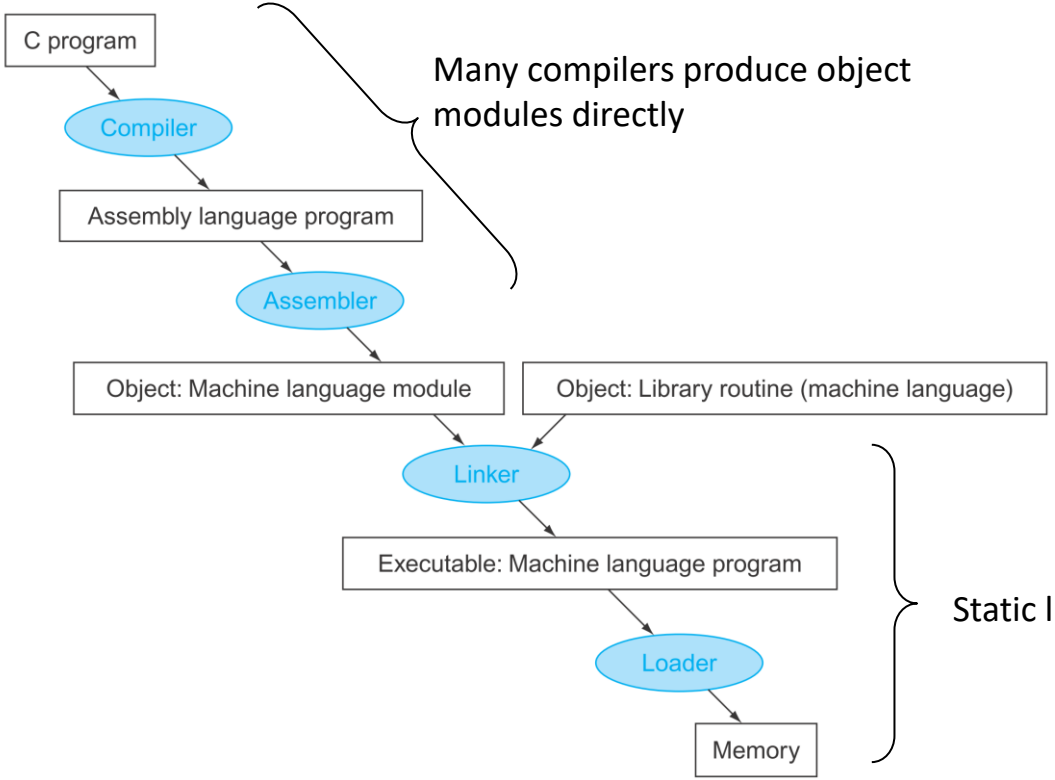
- Example 2: lock

```
      ADDI X11,XZR,#1        // copy locked value
again: LDXR X10,[X20,#0]     // load exclusive to read lock
      CBNZ X10, again        // check if it is 0 yet
      STXR X11, X9, [X20]    // attempt to store new value
      CBNZ X9,again          // branch if store fails
```

- Unlock:

```
      STUR XZR, [X20,#0]    // free lock by writing 0
```

Translating and Starting a Program



To identify the type of file, UNIX follows a suffix convention for files:
C source files are named x.c,
assembly files are x.s,
object files are named x.o,
statically linked library routines are x.a,
dynamically linked library routes are x.so,
executable files are called a.out.
MS-DOS uses the suffixes .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Compiler

- The compiler transforms the C program into an *assembly language program.*
- In 1975, many operating systems and assemblers were written in assembly language because memories were small and compilers were inefficient.
- Today compilers can produce assembly language programs nearly as well as an assembly language expert, and sometimes even better for large programs.

Assembler

- The assembler translates the assembly language program into machine language.
- It creates an **object file**, which is a combination of machine language instructions, data, and information needed to place instructions properly in memory.

- The assembler can also treat common variations of machine language instructions, i.e., **pseudoinstructions**. E.g., LEGv8 accepts

```
MOV X9,X10        // register X9 gets register X10
```

  - Converted to `ORR X9,XZR,X10 // register X9 gets 0 OR register X10`

```
CMP X9,X10  // compare X9 to X10 and set condition codes
```

  - Converted to `SUBS XZR,X9,X10  // use X9 – X10 to set condition codes`

```
AND X9,X10,#15 // register X9 gets X10 AND 15
```

  - Converted to `ANDI X9,X10,#15 // register X9 gets X10 AND 15`

Assembler

- To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels.
- Assemblers keep track of labels used in branches and data transfer instructions in a **symbol table**.
  - The table contains pairs of symbols and addresses.
- The **object file** for UNIX systems typically contains *six distinct pieces:*
  - The ***object file header*** describes the size and position of the other pieces of the object file.
  - The ***text segment*** contains the machine language code.
  - The ***static data segment*** contains data allocated for the life of the program.
  - The ***relocation information*** identifies instructions and data words that depend on absolute addresses when the program is loaded into memory.
  - The ***symbol table*** contains the remaining labels that are not defined, such as external references.
  - The ***debugging information*** contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files.

Linker (also called linker editor)

- The **linker** is a systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.
- There are three steps for the linker:
    1. Place code and data modules symbolically in memory.
    2. Determine the addresses of data and instruction labels.
    3. Patch both the internal and external references.
- The linker uses the relocation information and symbol table in each object module to resolve all undefined labels (i.e., in branch instructions and data addresses).
- If all external references are resolved, the linker next determines the memory locations each module will occupy.
- When the linker places a module in memory, all *absolute references*, that is, memory addresses that are not relative to a register, must be *relocated* to reflect its true location.
- The linker produces an **executable file** that can be run on a computer.
    - The **executable file** is a functional program in the format of an object file that contains no unresolved references.
    - It can contain symbol tables and debugging information.
    - Relocation information may be included for the loader.

Example: Linking Object Files

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	LDUR X0, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(X)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	LDUR	X
	4	BL	B
Symbol table	Label	Address	
	X	–	
	B	–	

Example: Linking Object Files

	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	STUR X1, [X27,#0]	
	4	BL 0	
	. . .	. . .	
Data segment	0	(Y)	
	. . .	. . .	
Relocation information	Address	Instruction type	Dependency
	0	STUR	Y
	4	BL	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Example: Linking Object Files

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0000\ 0000\ 0040\ 0000_{hex}$	LDUR X0, [X27,#0_{hex}]
	$0000\ 0000\ 0040\ 0004_{hex}$	BL 000 00FC$_{hex}$
	. . .	. . .
	$0000\ 0000\ 0040\ 0100_{hex}$	STUR X1, [X27,#20_{hex}]
	$0000\ 0000\ 0040\ 0104_{hex}$	BL 3FF FEFC$_{hex}$
	. . .	. . .
Data segment	Address	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	. . .	. . .
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	. . .	. . .

Loader

- The **loader** is a systems program that places an object program in main memory so that it is ready to execute.
- The loader follows these steps in UNIX systems:
  1. Reads the executable file header to determine size of the text and data segments.
  2. Creates an address space large enough for the text and data.
  3. Copies the instructions and data from the executable file into memory.
  4. Copies the parameters (if any) to the main program onto the stack.
  5. Initializes the processor registers and sets the stack pointer to the first free location.
  6. Branches to a start-up routine that copies the parameters into the argument registers and calls the main routine of the program. When the main routine returns, the start-up routine terminates the program with an exit system call.

Dynamically Linked Libraries

- We have described the traditional *static approach* to linking libraries before the program is run.
- It has a few disadvantages:
  1. The library routines become part of the executable code. If a new version of the library is released that fixes bugs or supports new hardware devices, the statically linked program keeps using the old version.
  2. It loads all routines in the library that are called anywhere in the executable, even if those calls are not executed. The library can be large relative to the program.
- These disadvantages lead to **dynamically linked libraries (DLLs)**, where the library routines are not linked and loaded until the program is run.
  - Both the program and library routines keep extra information on the location of nonlocal procedures and their names.
  - In the original version of DLLs, the loader ran a dynamic linker, using the extra information in the file to find the appropriate libraries and to update all external references.
  - But, it still linked all routines of the library that might be called.
  - In a more efficient approach each routine is linked only after it is called.

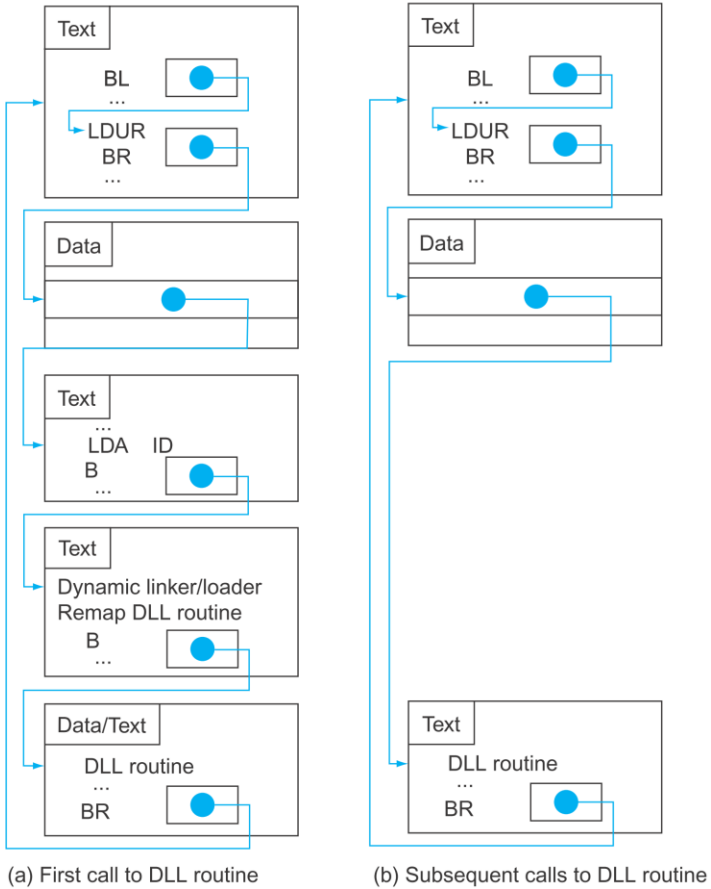
Dynamically Linked Libraries

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



A C Sort Example

- Swap procedure (leaf)

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

- v in X0, k in X1, temp in X9

A C Sort Example

```
swap: LSL X10,X1,#3        // X10 = k * 8
      ADD X10,X0,X10       // X10 = address of v[k]
      LDUR X9,[X10,#0]     // X9 = v[k]
      LDUR X11,[X10,#8]    // X11 = v[k+1]
      STUR X11,[X10,#0]    // v[k] = X11 (v[k+1])
      STUR X9,[X10,#8]     // v[k+1] = X9 (v[k])
      BR LR                // return to calling routine
```

A C Sort Example

- Sort procedure, non-leaf calls swap

```
void sort (long long int 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 X0, `n` in X1, `i` in X19, `j` in X20,
- we will also need to save `v` in X21, `n` in X22

A C Sort Example

- Skeleton of outer loop:

```
// for (i = 0; i <n; i += 1) {

        MOV X19,XZR                 // i = 0
for1tst:
        CMP X19, ~~X1~~ X22         // compare X19 to X1 (i to n)
        B.GE exit1                  // go to exit1 if X19 ≥ X1 (i≥n)

        (body of outer for-loop)

        ADDI X19,X19,#1             // i += 1
        B for1tst                   // branch to test of outer loop
exit1:
```

A C Sort Example

- Skeleton of inner loop:

```
//for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
        SUBI X20, X19, #1          // j = i - 1
for2tst: CMP X20,XZR               // compare X20 to 0 (j to 0)
        B.LT exit2                 // go to exit2 if X20 < 0 (j < 0)
        LSL X10, X20, #3           // reg X10 = j * 8
        ADD X11, X21, X10          // reg X11 = v + (j * 8)
        LDUR X12, [X11,#0]         // reg X12 = v[j]
        LDUR X13, [X11,#8]         // reg X13 = v[j + 1]
        CMP X12, X13               // compare X12 to X13
        B.LE exit2                 // go to exit2 if X12 ≤ X13
        MOV X0, X21                // first swap parameter is v
        MOV X1, X20                // second swap parameter is j
        BL swap                    // call swap
        MOV X1, X22                // needed for first loop comparison
        SUBI X20, X20, #1          // j -= 1
        B for2tst                  // branch to test of inner loop
exit2:
```

A C Sort Example

- Preserve saved registers:

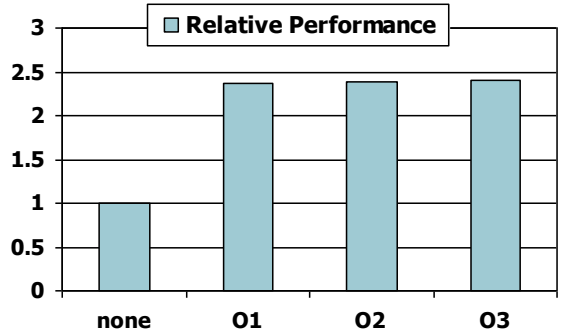
```
        SUBI SP,SP,#48    // make room on stack for 5 regs
        STUR LR,[SP,#32]  // save LR on stack
        STUR X22,[SP,#24] // save X22 on stack
        STUR X21,[SP,#16] // save X21 on stack
        STUR X20,[SP,#8]  // save X20 on stack
        STUR X19,[SP,#0]  // save X19 on stack
        MOV X21, X0                // copy parameter X0 into X21
        MOV X22, X1                // copy parameter X1 into X22
```

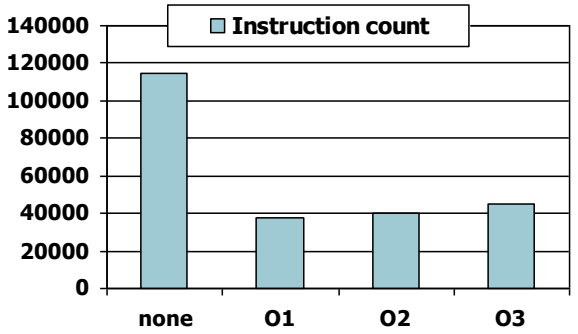
- Restore saved registers:

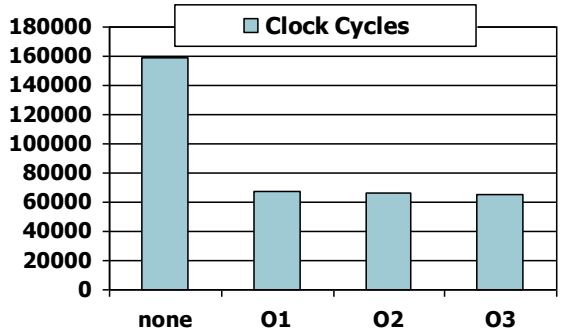
```
exit1: LDUR X19, [SP,#0]          // restore X19 from stack
       LDUR X20, [SP,#8]          // restore X20 from stack
       LDUR X21,[SP,#16]          // restore X21 from stack
       LDUR X22,[SP,#24]          // restore X22 from stack
       LDUR X30,[SP,#32]          // restore LR from stack
       ADDI SP,SP,#48             // restore stack pointer
       BR LR
```

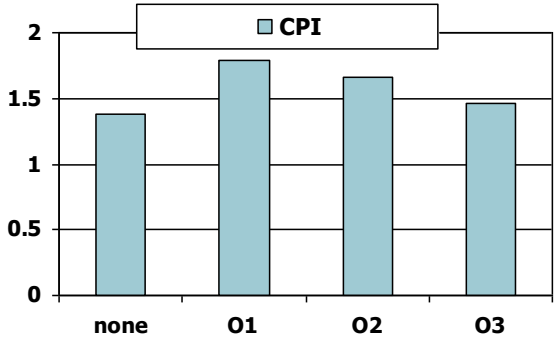
Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

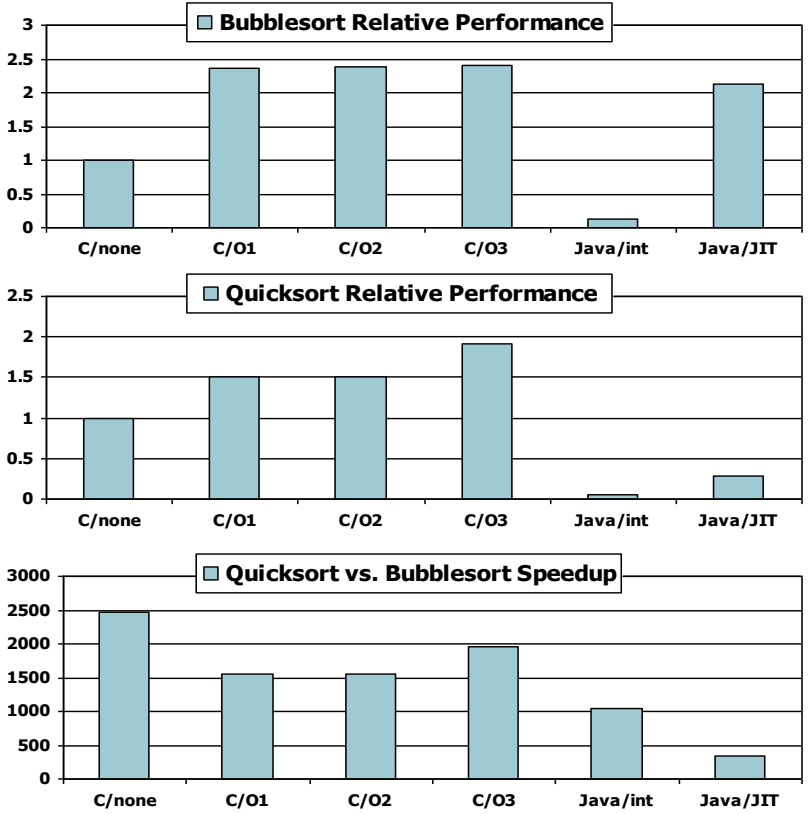








Effect of Language and Algorithm



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 versus Pointers

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

Arrays versus Pointers

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

```
      MOV X9,XZR      // i = 0
loop1: LSL X10,X9,#3  // X10 = i * 8
      ADD X11,X0,X10 // X11 = address
                     // of array[i]
      STUR XZR,[X11,#0]
                     // array[i] = 0
      ADDI X9,X9,#1  // i = i + 1
      CMP X9,X1      // compare i to
                     // size
      B.LT loop1     // if (i < size)
                     // go to loop1
```

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

```
      MOV X9,X0       // p = address of
                      // array[0]
      LSL X10,X1,#3   // X10 = size * 8
      ADD X11,X0,X10  // X11 = address
                      // of array[size]
loop2: STUR XZR,[X9,#0]
                     // Memory[p] = 0
      ADDI X9,X9,#8  // p = p + 8
      CMP X9,X11     // compare p to <
                     // &array[size]
      B.LT loop2     // if (p < &array[size])
                     // go to loop2
```

Arrays versus Pointers

- 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

Real Stuff: ARMv7 (32-bit) Instructions

- Standing originally for the **Acorn RISC Machine**, later changed to **Advanced RISC Machine.**
- ARMv1 came out in 1985 with 32 bit addresses.
- Many versions of the 32-bit address ARM instruction set came out over the years, culminating with ARMv7 in 2005.
- ARMv8 with 64-bit addresses was revealed in 2013, with big differences.

- Similarities between ARMv7 and ARMv8:
  - All instructions are 32 bits wide for both architectures.
  - The only way to access memory is via load and store instructions on both architectures.

- But …

Real Stuff: ARMv7 (32-bit) Instructions

- Here are some of the differences:
  - ARMv7 and the earlier ARM instruction sets have just 15 general-purpose registers.
  - No register is hardwired to 0, so ARMv7 and its predecessors need extra instructions to perform some operations that ARMv8 can do with XZR.
  - The missing 16th register in ARMv7 and its predecessors is the program counter (PC).
  - ARMv7 addressing modes do not work for all data sizes.
  - ARMv7 has Load Multiple and Store Multiple instructions. ARMv8 does not.
  - Rather than the immediate field simply being a constant, it is essentially an input to a function that produces a constant.
    - The eight least-significant bits of ARMv7's 12-bit immediate field are zero-extended to a 32-bit value and then rotated right the number of bits specified in the first four bits of the field multiplied by two.
  - Unlike ARMv8, the early ARM instruction sets omitted a divide instruction.

Real Stuff: The Rest of the ARMv8 Instruction set

Class	Loads/Stores		Operations		Branches		Total	
	AL	ML	AL	ML	AL	ML	AL	ML
Integer	49	145	74	105	–	–	123	250
Floating Point & Int Mul/Div	0	18	63	156	–	–	63	174
SIMD/Vector	16	166	229	371	–	–	245	537
System/Special	11	55	52	40	–	–	63	95
–	–	–	–	–	23	14	23	14
Total	76	384	418	672	23	14	517	1070

- Many assembly instruction are translated to different machine instructions (i.e. opcodes) according to the data the operate on.
- ARMv8 includes both 32-bit and 64-bit versions of instructions within the same architecture.
- In assembly language, programmers use registers named W0, W1, … instead of the X0, X1, … to specify 32-bit operations.

```
ADD    X9,X21,X9
ADD    W9,W21,W9
```

Real Stuff: The Rest of the ARMv8 Instruction set

- From ARMv8 Instruction Set overview:

- Most integer instructions in the A64 instruction set have two forms, which operate on either 32-bit or 64-bit values within the 64-bit general-purpose register file.
- Where a 32-bit instruction form is selected, the following holds true:
  - The upper 32 bits of the source registers are ignored;
  - The upper 32 bits of the destination register are set to ZERO;
  - Right shifts/rotates inject at bit 31, instead of bit 63;
  - The condition flags, where set by the instruction, are computed from the lower 32 bits.

ARMv8 Integer Arithmetic Logic Instructions

Type	Mnemonic	Instruction
Arithmetic Register	**ADD**	Add
	ADDS	Add and set flags
	SUB	Subtract
	SUBS	Subtract and set flags
	CMP	Compare
	CMN	Compare negative
	NEG	Negate
	NEGS	Negate and set flags
Arithmetic Immediate	**ADDI**	Add Immediate
	ADDIS	Add and set flags Immediate
	SUBI	Subtract Immediate
	SUBIS	Subtract and set flags Immediate
	CMPI	Compare Immediate
	CMNI	Compare negative Immediate
Arithmetic Extended	ADD	Add Extended Register
	ADDS	Add and set flags Extended
	SUB	Subtract Extended Register
	SUBS	Subtract and set flags Extended
	CMP	Compare Extended Register
	CMN	Compare negative Extended
Arithmetic with Carry	ADC	Add with carry
	ADCS	Add with carry and set flags
	SBC	Subtract with carry
	SBCS	Subtract with carry and set flags
	NGC	Negate with carry
	NGCS	Negate with carry and set flags
Logical Register	**AND**	Bitwise AND
	ANDS	Bitwise AND and set flags
	ORR	Bitwise inclusive OR
	EOR	Bitwise exclusive OR
	BIC	Bitwise bit clear
	BICS	Bitwise bit clear and set flags
	ORN	Bitwise inclusive OR NOT
	EON	Bitwise exclusive OR NOT
	MVN	Bitwise NOT
	TST	Test bits

Bold means the instruction is also in LEGv8, italic means it is a pseudoinstruction, and bold italic means it is a pseudoinstruction that is also in LEGv8.

ARMv8 Integer Arithmetic Logic Instructions

Type	Mnemonic	Instruction
Logical Immediate	**ANDI**	Bitwise AND Immediate
	ANDIS	Bitwise AND and set flags Immediate
	ORRI	Bitwise inclusive OR Immediate
	EORI	Bitwise exclusive OR Immediate
	TSTI	Test bits Immediate
Shift Register Shift Immed	**LSL**	Logical shift left Immediate
	LSR	Logical shift right Immediate
	ASR	Arithmetic shift right Immediate
	ROR	Rotate right Immediate
	LSRV	Logical shift right register
	LSLV	Logical shift left register
	ASRV	Arithmetic shift right register
	RORV	Rotate right register
Move Wide Immed iate	**MOVZ**	Move wide with zero
	MOVK	Move wide with keep
	MOVN	Move wide with NOT
	MOV	Move register

Bit Field Insert & Extract	BFM	Bitfield move
	SBFM	Signed bitfield move
	UBFM	Unsigned bitfield move (32-bit)
	BFI	Bitfield insert
	BFXIL	Bitfield extract and insert low
	SBFIZ	Signed bitfield insert in zero
	SBFX	Signed bitfield extract
	UBFIZ	Unsigned bitfield insert in zero
	UBFX	Unsigned bitfield extract
	EXTR	Extract register from pair
Sign Extend	*SXTB*	Sign-extend byte
	SXTH	Sign-extend halfword
	SXTW	Sign-extend word
	UXTB	Unsigned extend byte
	UXTH	Unsigned extend halfword
Bit Operation	CLS	Count leading sign bits
	CLZ	Count leading zero bits
	RBIT	Reverse bit order
	REV	Reverse bytes in register
	REV16	Reverse bytes in halfwords
	REV32	Reverses bytes in words

ARMv8 Integer Arithmetic Logic Instructions

- The second register of all arithmetic and logical processing operations has the option of being shifted before being operated on.
  - The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right.
  - Although the assembler has explicit instructions with these names (LSL, LSR, SRA, and ROR), these are really just pseudoinstructions.
- To support arithmetic on narrower data types, there are instructions that let you mix data sizes of the second operand by either sign extending it or zero extending it to the full width.
  - The *extended-register instructions* work with bytes, halfwords, or words.
- To support add and subtract operations on operands larger than one doubleword, ARM includes instructions to add or subtract the carry from a previous operation.
- **ASR** does arithmetic shift right, which replicates the sign bit during the shift, and **ROR** rotates the bits to the right; that is, the bits shift off to the right are inserted on the left.
- There are versions of all the shift instructions that determine the amount to be shifted based on a value in a register rather than as an immediate within the instruction.
- To manipulate fields of bits, the full ARMv8 instruction set includes instructions that can extract a bit field from a register and insert it into another.

ARMv8 Integer Data Transfer Instructions

- We did not see all of the addressing modes available, only the *unscaled signed immediate offset*.
- Here are five more:
  1. Base plus a scaled 12-bit unsigned immediate offset.
  2. Base plus a 64-bit register offset, optionally scaled.
  3. Base plus a 32-bit extended register offset, optionally scaled.
- The scaling options of the *first three addressing modes* multiply or scale the address in the immediate field or in the register by the size of the data being transferred in bytes.
  - Thus, if X11 contains $100{,}000_{ten}$

```
LDR X10, [X11, #16] // scaled addressing mode
```

  - will load the double word (8 bytes) at address $100{,}128_{ten}$ (100,000 + 8*16) into register X10.
- The address of the *second addressing mode* is simply the sum of two registers, with the option of shifting the second operand by 1, 2, or 3 bits
  - if X11 contains $100{,}000_{ten}$ and X12 contains $1{,}000_{ten}$

```
LDR X10, [X11, X12 LSL #3] // base + register, scaled
```

  - will load the double at address $108{,}000_{ten}$ (100,000 + 2<<3 *1000) into register X10.
- The third addressing mode simply uses a 32-bit register (e.g., W12) instead of a 64-bit register

ARMv8 Integer Data Transfer Instructions

4. Pre-indexed by an unscaled 9-bit signed immediate offset.
5. Post-indexed by an unscaled 9-bit signed immediate offset.

- These last two addressing modes *change the base register* as part of the address calculation.
- Thus, if X11 contains $100{,}000_{ten}$

```
LDR X10, [X11,#16]! // pre-indexed addressing mode
```

- will load the double word at address $100{,}0\underline{\mathbf{16}}_{ten}$ into register X10 and change X11 to $100{,}016_{ten}$.

```
LDR X10, [X11],#16 // post-indexed addressing mode
```

- will load the double word at address $100{,}0\underline{\mathbf{00}}_{ten}$ into register X10 and change X11 to $100{,}016_{ten}$.

- Among other things, to accelerate data transfers, ARMv8 includes three load pair and store pair instructions (**LDP, LDPSW, STP**), which transfer two doublewords at a time.

ARMv8 Integer Data Transfer Instructions

Type	Mnemonic	Instruction
Unscaled	**LDUR**	Load register (unscaled offset)
	LDURB	Load byte (unscaled offset)
	LDURSB	Load signed byte (unscaled offset)
	LDURH	Load halfword (unscaled offset)
	LDURSH	Load signed halfword (unscaled offset)
	LDURSW	Load signed word (unscaled offset)
	STUR	Store register (unscaled offset)
	STURB	Store byte (unscaled offset)
	STURH	Store halfword (unscaled offset)
	STURW	Store word (unscaled offset)
	LDA	Load address
Scaled, Extended, Pre-& Post-Indexed	LDR	Load register
	LDRB	Load byte
	LDRSB	Load signed byte
	LDRH	Load halfword
	LDRSH	Load signed halfword
	LDRSW	Load signed word
	STR	Store register
	STRB	Store byte
	STRH	Store halfword

Type	Mnemonic	Instruction
Exclusive	**LDXR**	Load Exclusive register
	LDXRB	Load Exclusive byte
	LDXRH	Load Exclusive halfword
	LDXP	Load Exclusive Pair
	STXR	Store Exclusive register
	STXRB	Store Exclusive byte
	STXRH	Store Exclusive halfword
	STXP	Store Exclusive Pair
Exclusive Aquire/Release	LDAXR	Load-aquire Exclusive register
	LDAXRB	Load-aquire Exclusive byte
	LDAXRH	Load-aquire Exclusive halfword
	LDAXP	Load-aquire Exclusive Pair
	STLXR	Store-release Exclusive register
	STLXRB	Store-release Exclusive byte
	STLXRH	Store-release Exclusive halfword
	STLXP	Store-release Exclusive Pair
Pair	LDP	Load Pair
	LDPSW	Load Pair signed words
	STP	Store Pair
PC	ADRP	Compute address of 4KB page at a PC-relative offset
	ADR	Compute address of label at a PC-relative offset

ARMv8 Branch Instructions

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Conditional Branch	**B.cond**	Branch conditionally	Conditional Select	CSEL	Conditional select
	CBNZ	Compare and branch if nonzero		CSINC	Conditional select increment
	CBZ	Compare and branch if zero		CSINV	Conditional select inversion
	TBNZ	Test bit and branch if nonzero		CSNEG	Conditional select negation
	TBZ	Test bit and branch if zero		*CSET*	Conditional set
Unconditional Branch	**B**	Branch unconditionally		*CSETM*	Conditional set mask
	BL	Branch with link		*CINC*	Conditional increment
	BLR	Branch with link to register		*CINV*	Conditional invert
	BR	Branch to register		*CNEG*	Conditional negate
	RET	Return from subroutine	Conditional Compare	CCMP	Conditional compare register
				CCMPI	Conditional compare immediate
				CCMN	Conditional compare negative register
				CCMNI	Conditional compare negative immediate

ARMv8 Branch Instructions

- There are two more unconditional branches:
  - The first is a variation of *branch and link* that uses a *register* for the branch address (**BLR**).
  - The second is *return from subroutine* (**RET**), which sounds a lot like branch register (BR);
    - The reason ARMv8 has different opcodes for the same operation is so that *hardware branch predictors can know* whether it is really return from a subroutine (RET), which is easy to predict, or being used in a branch table (BR), which is much harder to predict.

- There are instructions that store a value into a register based on the condition codes.
  - The idea behind *condition select instructions* is to replace conditional branches, which can cause problems in pipelined execution if they can't be predicted.

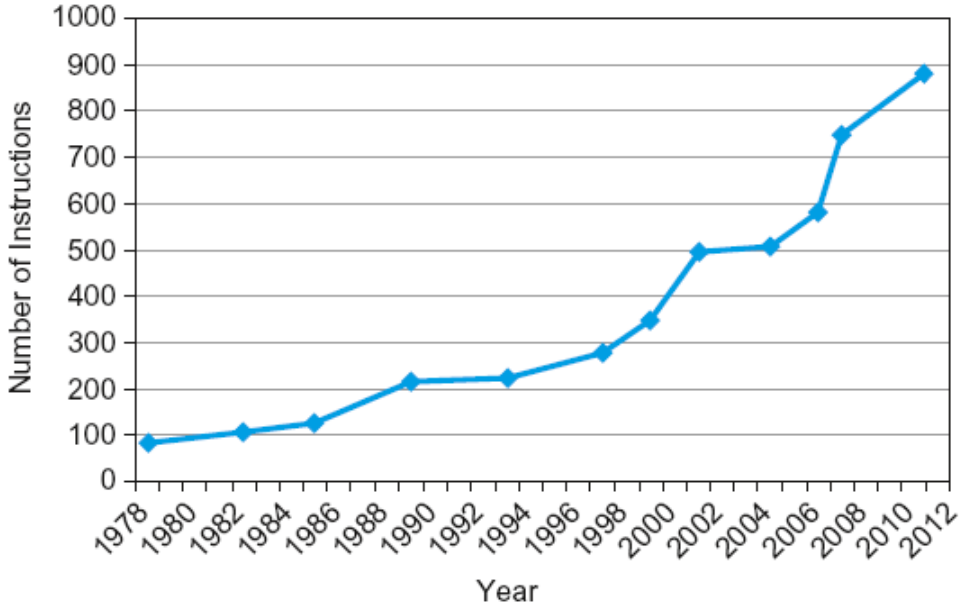
Fallacies

- 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

References

- David A. Patterson and John L. Hennessy, “Computer organization and design ARM edition: the hardware software interface,” Morgan Kaufmann, 2016.
- Chapter 2
  - Sections 2.1-2.3, 2.5-2.14, 2.17, 2.19, 2.20

Most of the text has been taken and adapted from “Computer Organization and Design ARM Edition: The Hardware Software Interface”.
If not differently indicated, all figures have been taken from the book or the material in the companion website of “Computer Organization and Design ARM Edition: The Hardware Software Interface”.