CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 14

Dr. Kinga Lipskoch

Fall 2019

Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudo-instructions'
  - e.g., `move $t0, $t1` exists only in assembly
  - would be implemented using `add $t0, $t1, $zero`
- When considering performance you should count real instructions

Array vs. Pointers (1)

```
1 void clear_1(int array[], int size) {
2   int i;
3   for (i = 0; i < size; i += 1)
4     array[i] = 0;
5 }
```

```
1 void clear_2(int *array, int size) {
2   int *p;
3   for (p = &array[0]; p < &array[size]; p++) {
4     *p = 0;
5 }
```

Array vs. Pointer (2)

Code assumes `size > 0`

Array

```
1           move $t0, $zero          # i = 0
2  LOOP1:   sll  $t1, $t0, 2         # $t1 = i * 4
3           add  $t2, $a0, $t1       # $t2 = &array[i]
4           sw   $zero, 0($t2)       # array[i] = 0
5           addi $t0, $t0, 1         # i++
6           slt  $t3, $t0, $a1       # $t3 = (i < size)
7           bne  $t3, $zero, LOOP1
```

Pointer

```
1           move $t0, $a0            # p = &array[0]
2           sll  $t1, $a1, 2         # $t1 = size * 4
3           add  $t2, $a0, $t1       # $t2 = &array[size]
4   LOOP2:  sw   $zero, 0($t0)       # Memory[p] = 0
5           addi $t0, $t0, 4         # p++ (addr += 4)
6           slt  $t3, $t0, $t2     # $t3=(p < &array[size])
7           bne  $t3, $zero, LOOP2
```

Array vs. Pointer (3)

- Array
  - multiply and add inside loop
  - address is recalculated from new index
- Pointer
  - increments pointer directly
  - less instructions inside loop
- Modern compilers might produce the same assembler code for both versions

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

  "The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions."

- Let's briefly look at IA-32

IA-32

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits + instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989 – 1995 The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new "MMX" instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and add more media extensions

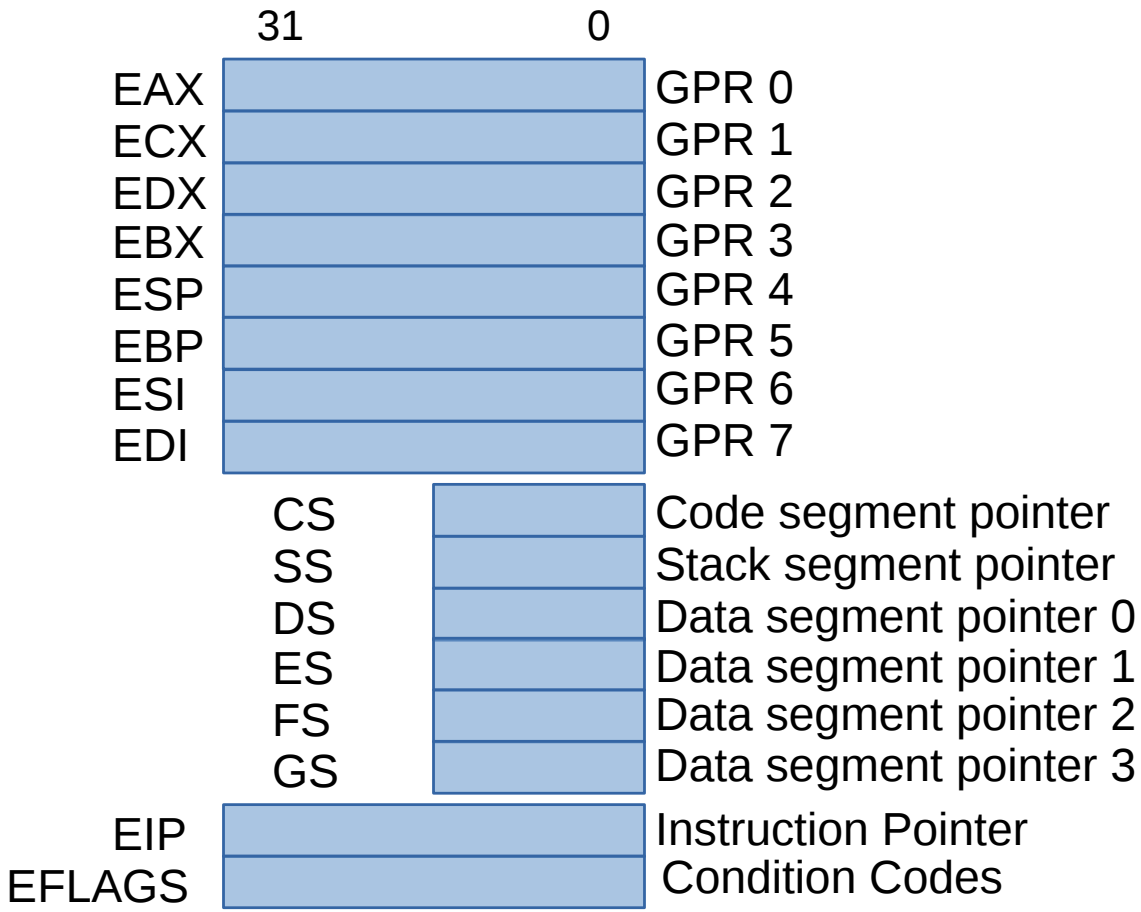
"This history illustrates the impact of the 'golden handcuffs' of compatibility"
"adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - One operand must act as both a source and destination
  - One operand can come from memory
  - Complex addressing modes, e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - The most frequently used instructions are not too difficult to build
  - Compilers avoid the portions of the architecture that are slow
  - "what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective"

IA-32 Addressing (80386)



IA-32 Register Restrictions

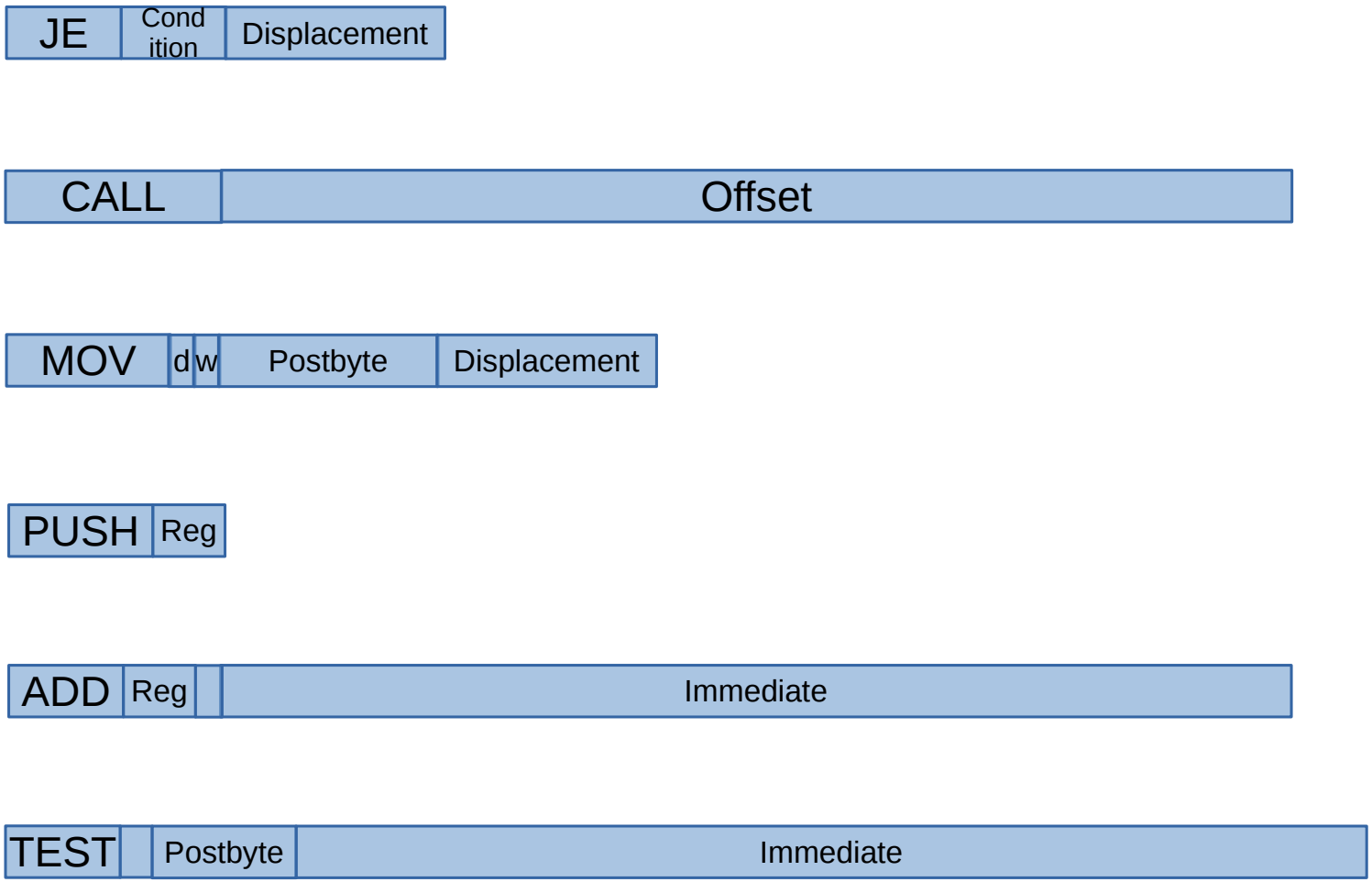
Registers are not "general purpose" – note the restrictions below

IA32 register restriction			
Mode	**Description**	**Register restrictions**	**MIPS equivalent**
Register indirect	Address is in a register	not ESP or EBP	`lw $0, 0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement	not ESP or EBP	`lw $s0, 100($s1) #≤16bit # displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base any GPR Index: not ESP	`mul $t0, $s2, 4`
			`add $t0, $t0, $s1`
			`lw $s0, 0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base any GPR Index: not ESP	`mul $t0, $s2, 4`
			`add $t0, $t0, $s1`
			`lw $s0, 100($t0) #≤16bit # displacement`

IA-32 Instruction Formats



IA-32 Typical Instructions

Four major types of integer instructions:

- Data movement including `move`, `push`, `pop`
- Arithmetic and logical (destination register or memory)
- Control flow (use of condition codes/flags)
- String instructions, including string move and string compare

IA-32 Instructions

IA-32 Instructions	
Instructions	**Meaning**
Control	**Conditional and unconditional branches**
JNZ, JZ	Jump if condition to EIP + 8- bit offset
JMP	Unconditionial jump – 8-bit or 16-bit offset
CALL	Subroutine call – 16-bit offset, return address pushed onto stack
RET	Pops return address from stack and jumps to it
LOOP	Loop branch – decrement ECX, jump to EIP + 8-bit displacement if ECX != 0
Data transfer	**Move data between registers or between register and memory**
MOV	Move between registers or between register and memory
PUSH, POP	Push source operand onto the stack, pop operand from stack top to a register
LES	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
ADD, SUB	add source to destination; subtract source from destination; register-memory format
CMP	Compare source and destination, register memory format
SHL, SHR, RCR	Shift left; shift logical right; rotate right with carry condition code as fill
CBW	Convert byte in 8 rightmost bits of EAX to 16-bit word in right of EAX
TEST	Logical AND of source and destination set condition codes
INC, DEC	Increment destination, decrement destination
OR, XOR	Logical OR; exclusive OR; register memory-format
String	**Move between string operands; length given by repeat prefix**
MOVS	Copies from string source to destination by incrementing ESI and EDI
LODS	Loads a byte, word or double of a string into EAX register

Summary

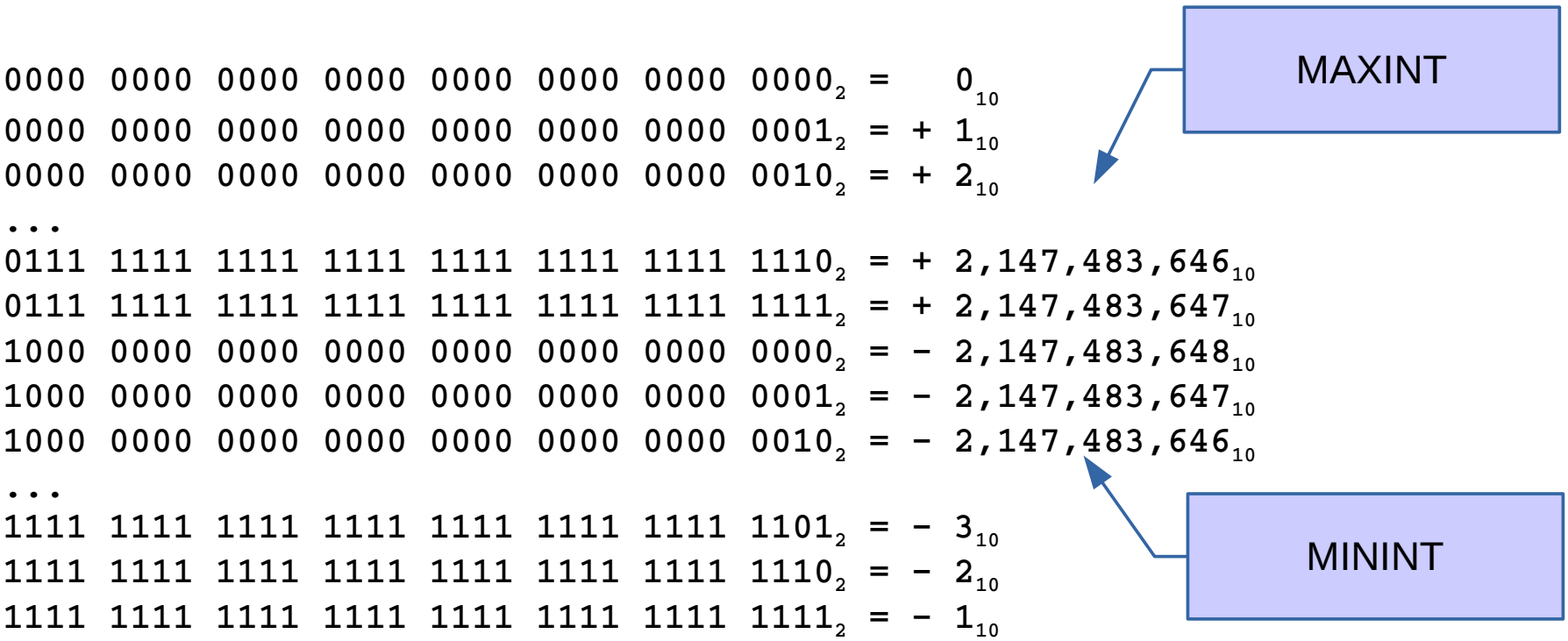
- Instruction complexity is only one variable
  - Lower instruction count vs. higher CPI/lower clock rate
- Design principles:
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands compromise
  - Make the common case fast
- Instruction set architecture
  - A very important abstraction

MIPS Tools

- Sourcery CodeBench Lite Edition:
  http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/editions/lite-edition/
- Allows to cross-compile MIPS code
- Inspect MIPS assembler:
  - `mips-linux-gnu-gcc -S -o example.s example.c`
- SPIM simulator:
  https://sourceforge.net/p/spimsimulator/code/HEAD/tree/

32 Bit Integers in MIPS

32 bit signed numbers:



Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction
- Overflow occurs if the value affects the sign:
  - Adding two positives yields a negative
  - Adding two negatives gives a positive
  - Subtract a negative from a positive and get a negative
  - Subtract a positive from a negative and get a positive

Effects of Overflow

- Details based on software system/language
  - C ignores integer overflow, Fortran requires program notification
  - Example: flight control vs. homework assignment
- Computer designer should provide way to detect overflow and ignore overflow
  - `add`, `addi`, `sub` cause exception
  - new MIPS instructions: `addu`, `addiu`, `subu` do not cause exception

    note: `addiu` still sign-extends

    note: `sltu`, `sltiu` for unsigned comparisons
- An exception (interrupt) occurs $\rightarrow$
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption

ALU: Arithmetic Logic Unit

- Brain of the computer
  - We have built a simple ALU that adds
- Traditional
  - Integer arithmetic (addition, subtraction)
    - (multiplication, division)
  - Support logic operations (and, nor, or, xor)
  - Bit-shifting
- Extended for MIPS:
  - Support the set-on-less-than instruction (`slt`)
  - Support test for equality (`bne`, `beq`)
    - Use subtraction: `(a - b) == 0` implies `a = b`
  - Detect overflow

Multiply for MIPS

- Separate pair of 32-bit registers to contain 64-bit product: `hi` and `lo`
- Two instructions: `mult`, `multu`
- Multiply produces a double precision product

```
mult $s0, $s1          # hi||lo = $s0 * $s1
```

  - Low-order word of the product is left in processor register `lo` and the high-order word is left in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

```
mflo $s1               # $s1 = lo
```

Division for MIPS

- Divide generates the remainder in `hi` and the quotient in `lo`

```
1 div $s0, $s1      # lo = $s0 / $s1
2                   # hi = $s0 mod $s1
```

  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and remainder to (user accessible) registers in the register file
- As with multiply, divide ignores overflow so software must determine if the quotient is too large
- Software must also check the divisor to avoid division by 0

Multiplication/Division

- Common hardware support for multiply and divide provides separate pair of 32-bit registers to contain 64-bit product or the remainder/quotient
- To fetch data from these `hi` and `lo` registers programmer uses `mflo` (move from low)
- MIPS multiply instructions ignore overflow, up to software to check. (No overflow if `hi` is 0), `mfhi` (move from hi) can be used to transfer `hi` to general-purpose register
- MIPS divide ignores overflow, up to software to check result and division by 0