

CS 110

Computer Architecture

MIPS Instruction Formats

Instructor:
Sören Schwertfeger

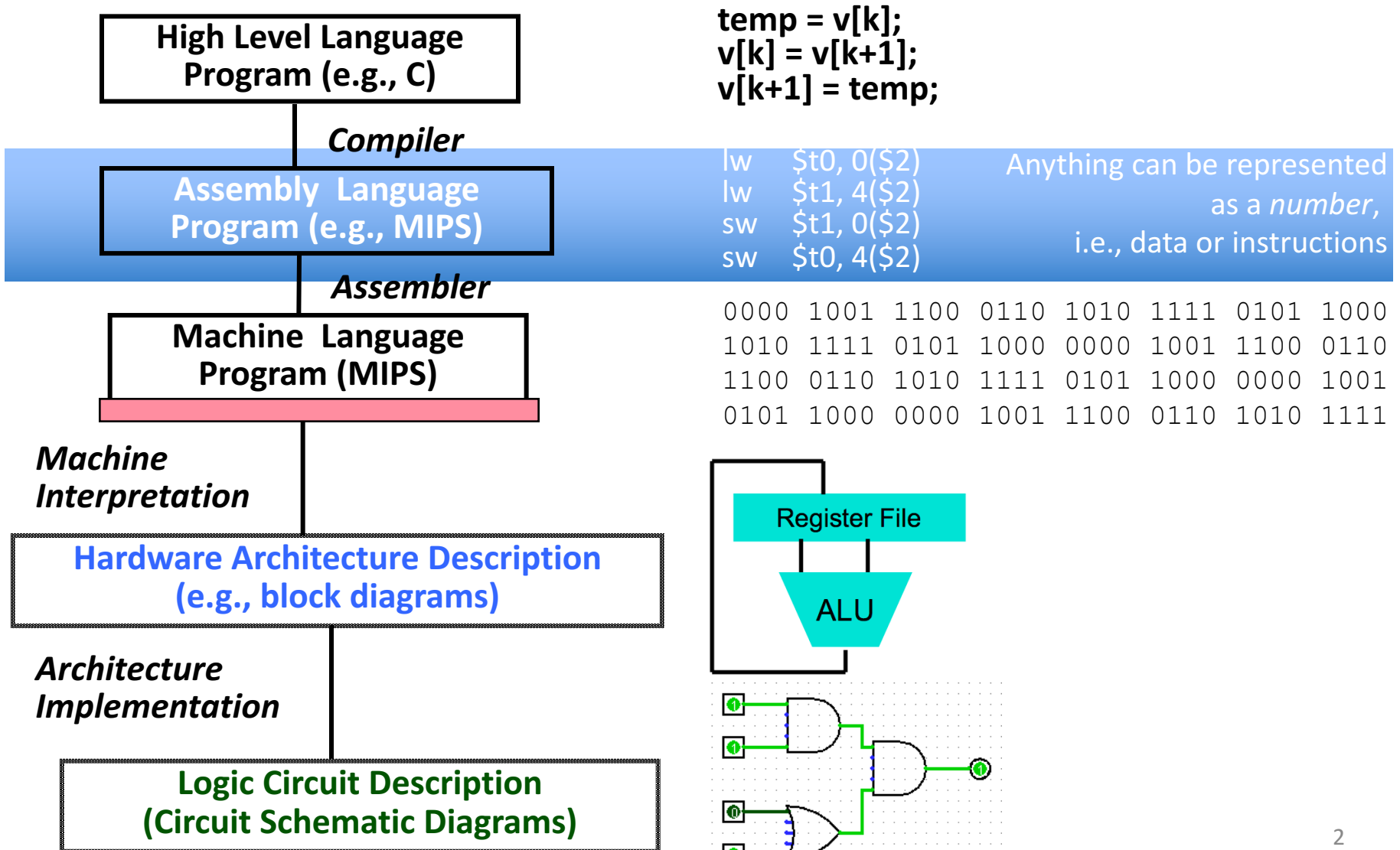
<http://shitech.org/courses/ca/>

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Levels of Representation/Interpretation



Review: Allocating Space on Stack

- C has two storage classes: automatic and static
 - *Automatic* variables are local to function and discarded when function exits
 - *Static* variables exist across exits from and entries to procedures
- Use stack for automatic (local) variables that don't fit in registers
- *Procedure frame* or *activation record*: segment of stack with saved registers and local variables
- Some MIPS compilers use a frame pointer ($\$fp$) to point to first word of frame

Stack Before, During, After Call

High address

\$fp →

\$sp →

Low address

a.

\$fp →

Saved argument
registers (if any)

Saved return address

Saved saved
registers (if any)

Local arrays and
structures (if any)

\$sp →

b.

\$fp →

\$sp →

c.

Recursive Function Factorial

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

Recursive Function Factorial

Fact:

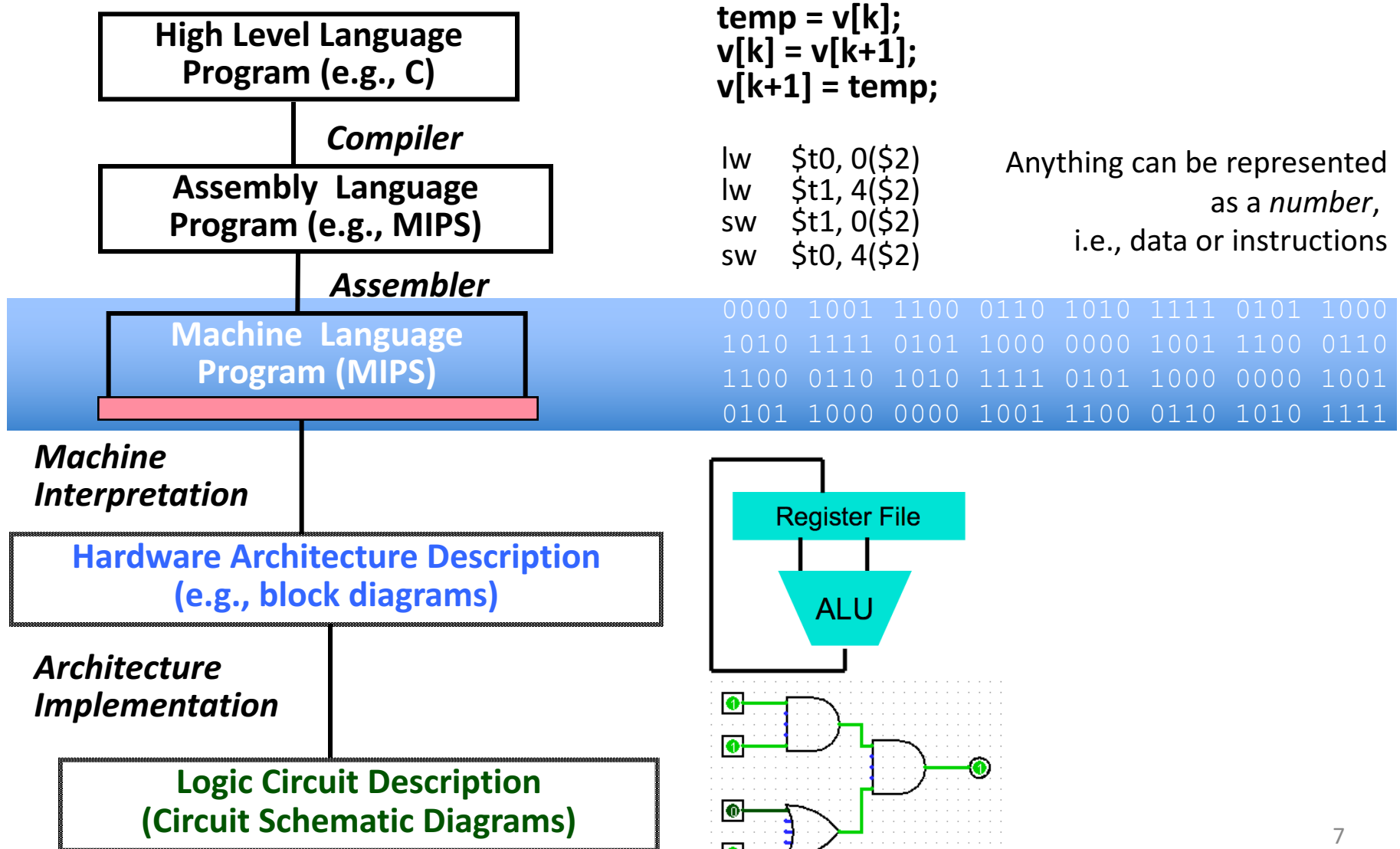
```
# adjust stack for 2 items
addi $sp,$sp,-8
# save return address
sw $ra, 4($sp)
# save argument n
sw $a0, 0($sp)
# test for n < 1
slti $t0,$a0,1
# if n >= 1, go to L1
beq $t0,$zero,L1
# Then part (n==1) return 1
addi $v0,$zero,1
# pop 2 items off stack
addi $sp,$sp,8
# return to caller
jr $ra
```

L1:

```
# Else part (n >= 1)
# arg. gets (n - 1)
addi $a0,$a0,-1
# call fact with (n - 1)
jal Fact
# return from jal: restore n
lw $a0, 0($sp)
# restore return address
lw $ra, 4($sp)
# adjust sp to pop 2 items
addi $sp, $sp, 8
# return n * fact (n - 1)
mul $v0,$a0,$v0
# return to the caller
jr $ra
```

mul is a pseudo instruction

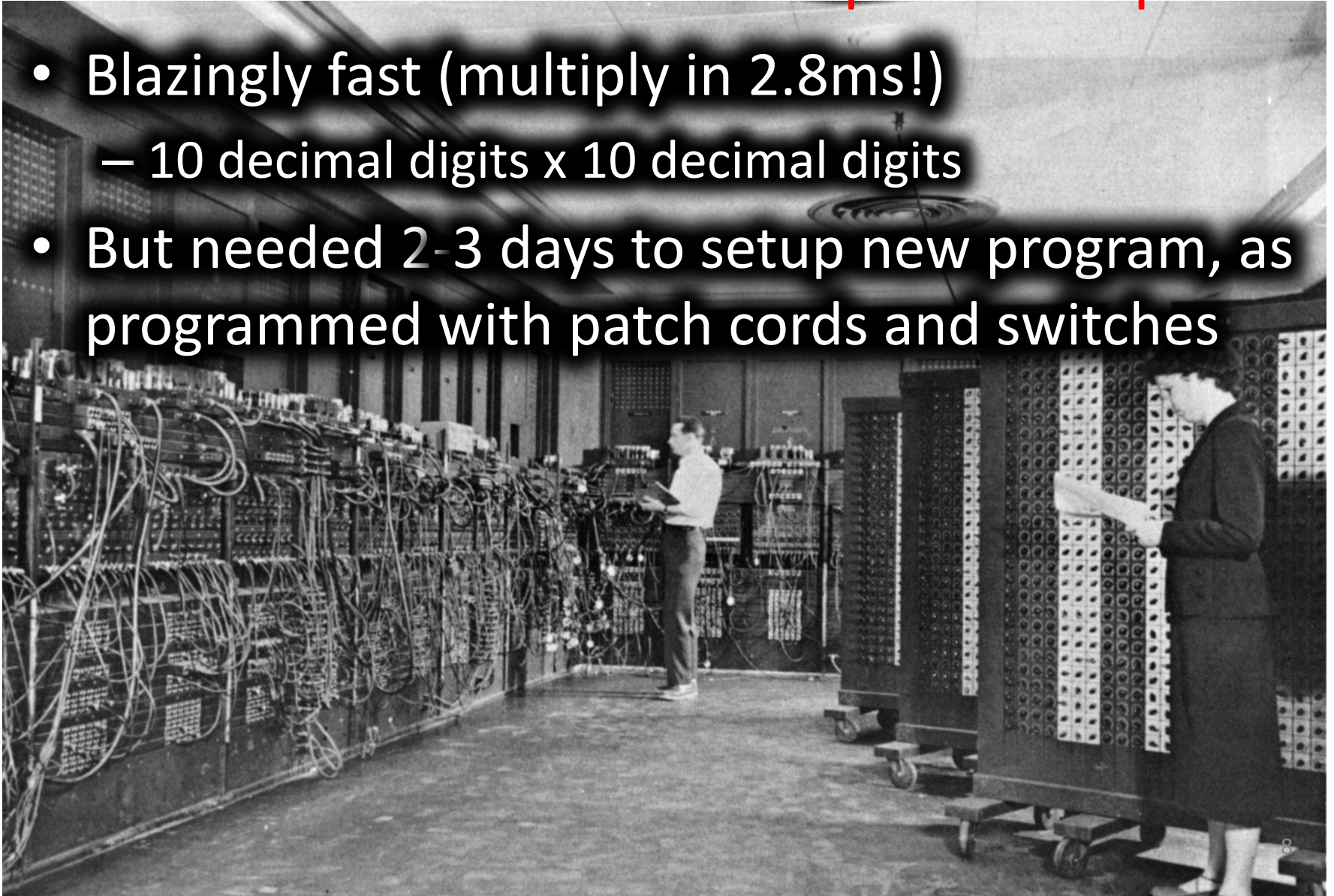
Levels of Representation/Interpretation



ENIAC (U.Penn., 1946)

First Electronic General-Purpose Computer

- Blazingly fast (multiply in 2.8ms!)
 - 10 decimal digits x 10 decimal digits
- But needed 2-3 days to setup new program, as programmed with patch cords and switches



Big Idea: Stored-Program Computer

First Draft of a Report on the EDVAC
by
John von Neumann
Contract No. W-670-ORD-4926
Between the
United States Army Ordnance Department and the
University of Pennsylvania
Moore School of Electrical Engineering
University of Pennsylvania
June 30, 1945

- Instructions are represented as bit patterns - can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don't have to rewire computer (days)
- Known as the “von Neumann” computers after widely distributed tech report on EDVAC project
 - Wrote-up discussions of Eckert and Mauchly
 - Anticipated earlier by Turing and Zuse

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
 - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
 - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limited in Java by language design
- One register keeps address of instruction being executed: **“Program Counter” (PC)**
 - Basically a pointer to memory: Intel calls it Instruction Pointer (a better name)

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
 - Programs bound to specific instruction set
 - Different version for **Macintoshes** and **PCs**
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to “backward-compatible” instruction set evolving over time
- Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit chunks):
 - Each register is a word.
 - **lw** and **sw** both access memory one word at a time.
- So how do we represent instructions?
 - Remember: Computer only understands 1s and 0s, so “**add \$t0, \$0, \$0**” is meaningless.
 - MIPS/RISC seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into “**fields**”.
- Each field tells processor something about instruction.
- We could define different fields for each instruction, but MIPS seeks simplicity, so define 3 basic types of instruction formats:
 - R-format
 - I-format
 - J-format

Instruction Formats

- **I-format**: used for instructions with immediates, **lw** and **sw** (since offset counts as an immediate), and branches (**beq** and **bne**)
 - (but not the shift instructions; later)
- **J-format**: used for **j** and **jal**
- **R-format**: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way

R-Format Instructions (1/5)

- Define “**fields**” of the following number of bits each: $6 + 5 + 5 + 5 + 5 + 6 = 32$

6	5	5	5	5	6
---	---	---	---	---	---

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct
---------------	-----------	-----------	-----------	--------------	--------------

- Important:** On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer
 - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63

R-Format Instructions (2/5)

- What do these field integer values tell us?
 - **opcode**: partially specifies what instruction it is
 - Note: This number is equal to **0** for all R-Format instructions
 - **funct**: combined with **opcode**, this number exactly specifies the instruction
- Question: Why aren't **opcode** and **funct** a single 12-bit field?
 - We'll answer this later

R-Format Instructions (3/5)

- More fields:
 - rs (Source Register): *usually* used to specify register containing first operand
 - rt (Target Register): *usually* used to specify register containing second operand (note that name is misleading)
 - rd (Destination Register): *usually* used to specify register which will receive result of computation

R-Format Instructions (4/5)

- Notes about register fields:
 - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
 - The word “*usually*” was used because there are exceptions that we’ll see later

R-Format Instructions (5/5)

- Final field:
 - shamt: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31)
 - This field is set to 0 in all but the shift instructions
- For a detailed description of field usage for each instruction, see green insert in COD (You can bring with you to all exams)

R-Format Example (1/2)

- MIPS Instruction:

add \$8 , \$9 , \$10

opcode = 0 (look up in table in book)

funct = 32 (look up in table in book)

rd = 8 (destination)

rs = 9 (first *operand*)

rt = 10 (second *operand*)

shamt = 0 (not a shift)

R-Format Example (2/2)

- MIPS Instruction:

add \$8, \$9, \$10

Decimal number per field representation:

0	9	10	8	0	32
---	---	----	---	---	----

Binary number per field representation:

000000	01001	01010	01000	00000	100000
--------	-------	-------	-------	-------	--------

hex

hex representation: $012A\ 4020_{\text{hex}} = 19,546,144_{\text{ten}}$

Called a Machine Language Instruction

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

I-Format Instructions (1/4)

- What about instructions with immediates?
 - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
 - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
 - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/4)

- Define “fields” of the following number of bits each:
 $6 + 5 + 5 + 16 = 32$ bits

6	5	5	16
---	---	---	----

- Again, each field has a name:

opcode	rs	rt	immediate
---------------	-----------	-----------	------------------

- **Key Concept:** Only one field is inconsistent with R-format. Most importantly, **opcode** is still in same location.

I-Format Instructions (3/4)

- What do these fields mean?
 - **opcode**: same as before except that, since there's no **funct** field, **opcode** uniquely specifies an instruction in I-format
 - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent as possible with other formats while leaving as much space as possible for immediate field.
 - **rs**: specifies a register operand (if there is one)
 - **rt**: specifies register which will receive result of computation (this is why it's called the *target* register "**rt**") or other operand for some instructions.

I-Format Instructions (4/4)

- The Immediate Field:
 - **addi**, **slti**, **sltiu**, the immediate is **sign-extended** to 32 bits. Thus, it's treated as a signed integer.
 - 16 bits → can be used to represent immediate up to 2^{16} different values
 - This is large enough to handle the offset in a typical **lw** or **sw**, plus a vast majority of values that will be used in the **slti** instruction.
 - Later, we'll see what to do when a value is too big for 16 bits

I-Format Example (1/2)

- MIPS Instruction:

addi \$21 , \$22 , -50

opcode = 8 (look up in table in book)

rs = 22 (register containing operand)

rt = 21 (target register)

immediate = -50 (by default, this is decimal in assembly code)

I-Format Example (2/2)

- MIPS Instruction:

`addi $21 , $22 , -50`

Decimal/field representation:

8	22	21	-50
---	----	----	-----

Binary/field representation:

001000	10110	10101	1111111111001110
--------	-------	-------	------------------

hexadecimal representation: 22D5 FFCE_{hex}

Question

Which instruction has same representation as integer 35_{ten} ?

a) add \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

b) subu \$s0,\$s0,\$s0

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

c) lw \$0, 0(\$0)

opcode	rs	rt	offset		
--------	----	----	--------	--	--

d) addi \$0, \$0, 35

opcode	rs	rt	immediate		
--------	----	----	-----------	--	--

e) subu \$0, \$0, \$0

opcode	rs	rt	rd	shamt	funct
--------	----	----	----	-------	-------

Registers numbers and names:

0: \$0, .. 8: \$t0, 9:\$t1, ..15: \$t7, 16: \$s0, 17: \$s1, .. 23: \$s7

Opcodes and function fields:

add: opcode = 0, funct = 32

subu: opcode = 0, funct = 35

addi: opcode = 8

lw: opcode = 35

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
 - Sometimes want to use immediates $> \pm 2^{15}$ with `addi`, `lw`, `sw` and `slti`
 - Bitwise logic operations with 32-bit immediates
- **Solution:** Don't mess with instruction formats, just add a new instruction
- **Load Upper Immediate** (`lui`)
 - `lui reg, imm`
 - Moves 16-bit `imm` into upper half (bits 16-31) of `reg` and zeros the lower half (bits 0-15)

lui Example

- Want: `addiu $t0, $t0, 0xABABCD`
– This is a **pseudo-instruction**!

- Translates into:

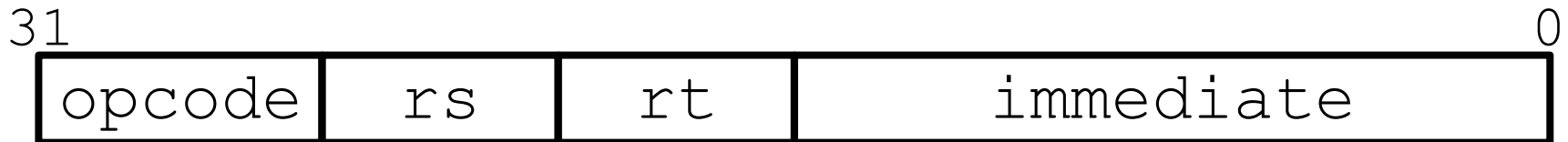
```
lui    $at, 0xABAB      # upper 16
ori    $at, $at, 0xCDCD # lower 16
addu   $t0, $t0, $at     # move
```

 Only the assembler gets to use \$at (\$1)

- Now we can handle everything with a 16-bit immediate!

Branching Instructions

- `beq` and `bne`
 - Need to specify a target address if branch taken
 - Also specify two registers to compare
- Use I-Format:



- opcode specifies `beq` (4) vs. `bne` (5)
- `rs` and `rt` specify registers
- How to best use `immediate` to specify addresses?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
 - Loops are generally small (< 50 instructions)
 - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
 - Largest branch distance limited by size of code
 - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- **PC-Relative Addressing:** Use the `immediate` field as a two's complement offset to PC
 - Branches generally change the PC by a small amount
 - Can specify $\pm 2^{15}$ addresses from the PC

Branch Calculation

- If we **don't** take the branch:
 - $PC = PC + 4 = \text{next instruction}$
- If we **do** take the branch:
 - $PC = (PC+4) + (\text{immediate} * 4)$
- **Observations:**
 - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)
 - Branch from $PC+4$ for hardware reasons; will be clear why later in the course

Branch Example (1/2)

- MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $9, $9, -1
      j      Loop
End:
```



- I-Format fields:

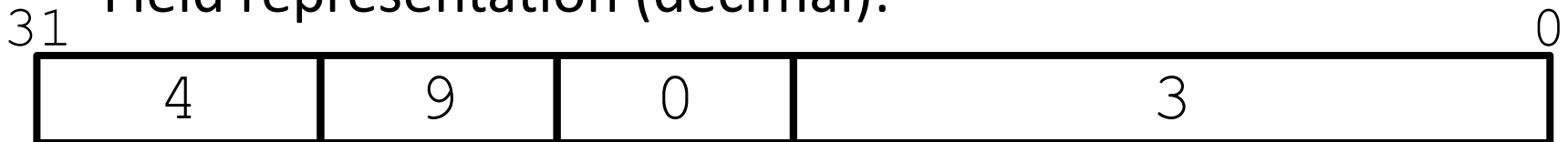
opcode = 4	(look up on Green Sheet)
rs = 9	(first operand)
rt = 0	(second operand)
immediate = 3	

Branch Example (2/2)

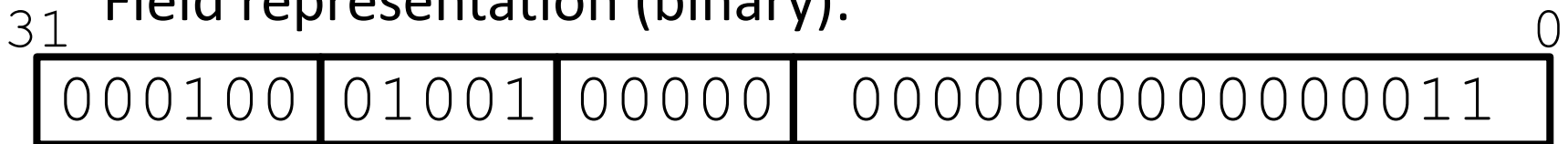
- MIPS Code:

```
Loop: beq    $9, $0, End
      addu   $8, $8, $10
      addiu  $9, $9, -1
      j      Loop
End:
```

Field representation (decimal):



Field representation (binary):



Questions on PC-addressing

- [illegible]

J-Format Instructions (1/4)

- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`j` and `j al`), we may jump to *anywhere* in memory
 - Ideally, we would specify a 32-bit memory address to jump to
 - Unfortunately, we can't fit both a 6-bit `opcode` and a 32-bit address into a single 32-bit word

J-Format Instructions (2/4)

- Define two “fields” of these bit widths:



- As usual, each field has a name:



- Key Concepts:**
 - Keep `opcode` field identical to R-Format and I-Format for consistency
 - Collapse all other fields to make room for large target address

J-Format Instructions (3/4)

- We can specify 2^{26} addresses
 - Still going to word-aligned instructions, so add 00 as last two bits (multiply by 4)
 - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
 - Cannot reach *everywhere*, but adequate almost all of the time, since programs aren't that long
 - Only problematic if code straddles a 256MB boundary
- If necessary, use 2 jumps or `jr` (R-Format) instead

J-Format Instructions (4/4)

- Jump instruction:
 - New PC = { (PC+4)[31..28], target address, 00 }
- Notes:
 - { , , } means concatenation
{ 4 bits , 26 bits , 2 bits } = 32 bit address
 - Book uses || instead
 - Array indexing: [31..28] means highest 4 bits
 - For hardware reasons, use PC+4 instead of PC

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
 - e.g. assignment ($a=b$) via `add $zero`
- MIPS has a set of “pseudo-instructions” to make programming easier
 - More intuitive to read, but get translated into actual instructions later

- Example:

```
move dst, src
```

translated into

```
addi dst, src, 0
```

Assembler Pseudo-Instructions

- List of pseudo-instructions:
http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
 - List also includes instruction translation
- **Load Address** (`la`)
 - `la dst, label`
 - Loads address of specified label into `dst`
- **Load Immediate** (`li`)
 - `li dst, imm`
 - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
 - See Help (F1) for full list

Assembler Register

- Problem:
 - When breaking up a pseudo-instruction, the assembler may need to use an extra register
 - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
 - Reserve a register (**\$1** or **\$at** for “assembler temporary”) that assembler will use to break up pseudo-instructions
 - Since the assembler may use this at any time, it's not safe to code with it

Multiply and Divide

- Example pseudo-instruction:

```
mul $rd,$rs,$rt
```

- Consists of **mult** which stores the output in special **hi** and **lo** registers, and a move from these registers to **\$rd**

```
mult $rs,$rt
```

```
mflo $rd
```

- **mult** and **div** have nothing important in the **rd** field since the destination registers are **hi** and **lo**
- **mfhi** and **mflo** have nothing important in the **rs** and **rt** fields since the source is determined by the instruction (see COD)

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

Multiplicand 1000 8 Multiplier
x1001 9

```
      1000
      0000
      0000
+ 1000
-----
01001000
```

- m bits \times n bits = $m + n$ bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
 - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
 - `mult register1, register2`
 - Multiplies 32-bit values in those registers & puts 64-bit product in special result regs:
 - puts product upper half in `hi`, lower half in `lo`
 - `hi` and `lo` are 2 registers separate from the 32 general purpose registers
 - Use `mghi` register & `mflo` register to move from `hi`, `lo` to another register

Integer Multiplication (3/3)

- Example:

- in C: `a = b * c;`

- in MIPS:

- let `b` be `$s2`; let `c` be `$s3`; and let `a` be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult  $s2,$s3    # b*c
mfhi  $s0         # upper half of
                # product into $s0
mflo  $s1         # lower half of
                # product into $s1
```

- Note: Often, we only care about the lower half of the product.
- Pseudo-inst. `mul` expands to `mult/mflo`.

Integer Division (1/2)

- Paper and pencil example (unsigned):

```

      1001   Quotient
1000 | 1001010 Dividend
     -1000
     -----
        10
        101
        1010
        -1000
        -----
           10   Remainder
                (or Modulo result)
```

- $\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$

Integer Division (2/2)

- Syntax of Division (signed):
 - `div` register1, register2
 - Divides 32-bit register 1 by 32-bit register 2:
 - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (/) and modulo (%)
- Example in C: `a = c / d;` `b = c % d;`
- in MIPS: `a↔$s0; b↔$s1; c↔$s2; d↔$s3`

```
div    $s2,$s3      # lo=c/d, hi=c%d
mflo   $s0           # get quotient
mfhi   $s1           # get remainder
```

MAL vs. TAL

- True Assembly Language (TAL)
 - The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
 - Instructions the assembly programmer can use (includes pseudo-instructions)
 - Each MAL instruction becomes 1 or more TAL instruction

Question

Which of the following place the address of LOOP in \$v0?

1) `la $t1, LOOP`
`lw $v0, 0($t1)`

2) `jal LOOP`
`LOOP: addu $v0, $ra, $zero`

3) `la $v0, LOOP`

	1	2	3
A)	T	T	T
B)	T	T	F
C)	F	T	T
D)	F	T	F
E)	F	F	T

Summary

- **I-Format:** instructions with immediates, `lw/sw` (offset is immediate), and `beq/bne`
 - But not the shift instructions
 - Branches use PC-relative addressing



- **J-Format:** `j` and `jal` (but not `jr`)
 - Jumps use absolute addressing



- **R-Format:** all other instructions

