Instruction Representation

Review (1/2)

- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word
  - Use to isolate fields, either by masking or by shifting back and forth
  - Shift left logical (`sll`) multiplies by powers of 2
  - Shift right arithmetic (`sra`) divides by powers of 2
    close but strange rounding for negative numbers
    (e.g., -5 `sra` 2 bits = -2 *while* -5 / 2^2 = -5 / 4 = -1)
- New Instructions:
  `and andi or ori nor sll srl sra`

Review (2/2)

- MIPS Signed versus Unsigned is an "overloaded" term
  - Do/Don't sign extend (lb, lbu)
  - Don't overflow (addu, addiu, subu)
  - Compute the correct answer (multu, divu)
  - Do signed/unsigned compare (slt,slti/sltu,sltiu)

Big Idea: Stored-Program Concept

- Computers built on 2 key principles:
  1) Instructions are represented as binary data.
  2) Therefore, entire programs can be stored in memory to be read or written just like binary data.
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory as Binary data, everything has a memory address
  - Instruction words and data words
  - Both branches and jumps use these instruction addresses
- C pointers are just memory addresses: can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs!
  - Up to you in MIPS; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Just a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
- New machines want to run old programs (“binaries”) as well as programs compiled to new instructions
- Leads to instruction set evolving over time
- Intel 8086 was selected in 1981 for 1st IBM PC
  - Latest PCs still use 80x86 instruction set…
  - Can (more or less) still run program from 1981 PC today!


0x28800001
0x1100fffe
0x20020001
0x23bd0008
0x03e00008

Instructions as Binary Data (1/2)

- All data we work with is in words (32-bit blocks):
  - 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 wants simplicity:
    - Since data is in words, let the instructions be words too

Instructions as Numbers (2/2)

- Divide the 32-bit instruction word into “fields”
- Each field tells something about the instruction
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - R-format
  - I-format
  - J-format (next lecture)

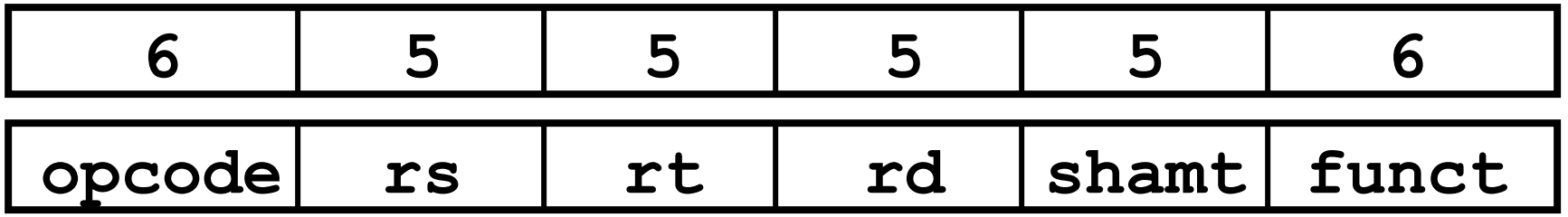
Recall Different Instructions

Category	Instruction	Example		Meaning	Comments
Arithmetic	add	add	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu	$s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu	$s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu	$s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0	$s1,$epc	$s1 = $epc	Used to copy Exception PC plus other special registers
	multiply	mult	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu	$s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo
	divide	div	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Lo = quotient, Hi = remainder
	divide unsigned	divu	$s2,$s3	Lo = $s2 / $s3, Hi = $s2 mod $s3	Unsigned quotient and remainder
	move from Hi	mfhi	$s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo	$s1	$s1 = Lo	Used to get copy of Lo
Logical	and	and	$s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; logical AND
	or	or	$s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; logical OR
	and immediate	andi	$s1,$s2,100	$s1 = $s2 & 100	Logical AND reg, constant
	or immediate	ori	$s1,$s2,100	$s1 = $s2 | 100	Logical OR reg, constant
	shift left logical	sll	$s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl	$s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Data transfer	load word	lw	$s1,100($s2)	$s1 = Memory[$s2+100]	Word from memory to register
	store word	sw	$s1,100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte unsigned	lbu	$s1,100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb	$s1,100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui	$s1,100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq	$s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne	$s1,$s2,25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; two's complement
	set less than immediate	slti	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1=0	Compare < constant; two's complement
	set less than unsigned	sltu	$s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1=0	Compare less than; natural numbers
	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare < constant; natural numbers
Unconditional jump	jump	j	2500	go to 10000	Jump to target address
	jump register	jr	$ra	go to $ra	For switch, procedure return
	jump and link	jal	2500	$ra = PC + 4; go to 10000	For procedure call

Instruction Formats

- I-format: used for instructions with *immediates*,
  - **`lw`** and **`sw`** (since the offset counts as an immediate),
  - **`beq`** and **`bne`** (branches use offsets as we will see later)
  - But not the shift instructions (more on this later)
- J-format: *jump* format used for **`j`** and **`jal`**
- R-format: used for all other instructions
  - R stands for *register* format
- It will soon become clear why the instructions have been partitioned in this way!

R-Format Instructions (1/5)



- Break 32 bit "instruction" word into fields
  - For simplicity each field has a name
- **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

  5 bit fields can represent any number 0-31,

  6 bit fields can represent any number 0-63.

R-Format Instructions (2/5)

6	5	5	5	5	6
opcode	**rs**	**rt**	**rd**	**shamt**	**funct**

- What do these field integer values tell us?

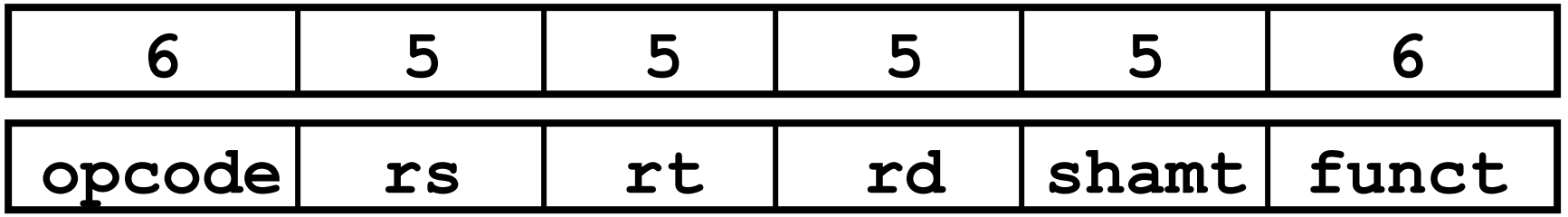
  opcode partially specifies what instruction it is

  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?
    - Think about it… We'll see the answer this later.

R-Format Instructions (3/5)



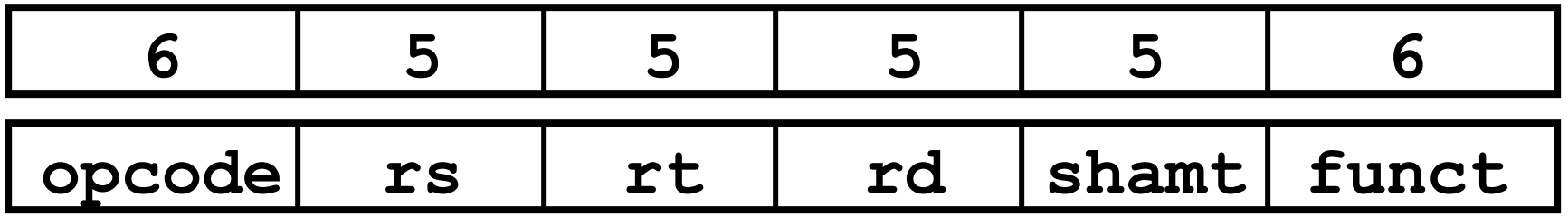
- More fields

  `rs` (Source Register): ***generally*** used to specify register containing **first operand**

  `rt` (Target Register): ***generally*** used to specify register containing **second operand** (note that name is misleading)

  `rd` (Destination Register): ***generally*** 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
  - It can specify any unsigned integer in the range 0-31
  - It specifies one of the 32 registers by number
- "**generally**" on previous slide because there are exceptions that we'll discuss more later…

R-Format Instructions (5/5)

6	5	5	5	5	6
opcode	rs	rt	rd	shamt	funct

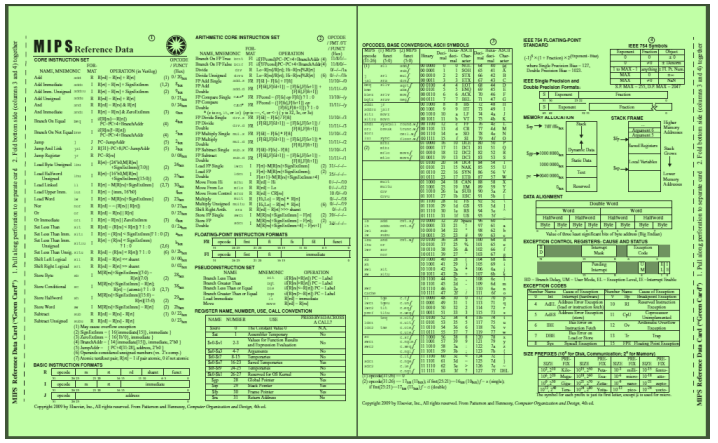
- One more field we have not yet discussed

  `shamt` contains the amount a shift instruction will shift

  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 reference card in the textbook



R-Format Example (1/2)

- MIPS Instruction:

```
add    $8 $9 $10
```

opcode = 0 (look up in table)

funct = 32 (look up in table)

rs = 9 (first *operand*)

rt = 10 (second *operand*)

rd = 8 (destination)

shamt = 0 (not a shift)

MIPS Reference Data ①

CORE INSTRUCTION SET

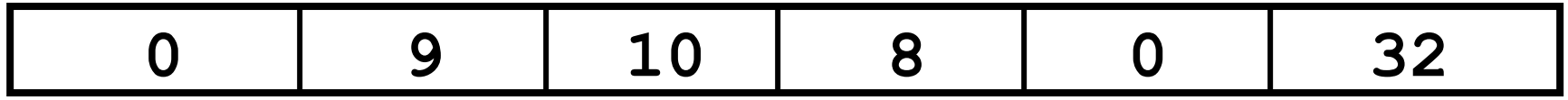
NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	or	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}

R-Format Example (2/2)

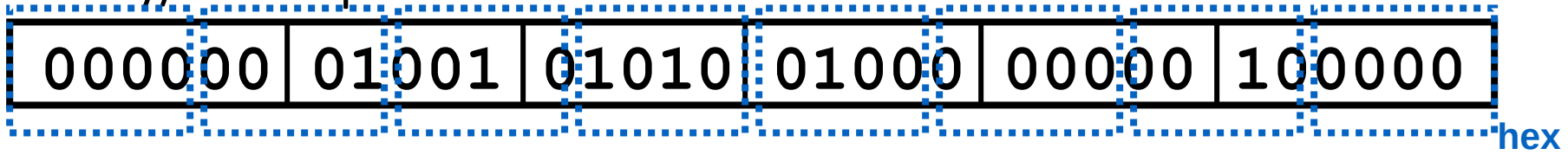
- MIPS Instruction:

```
add   $8 $9 $10
```

Decimal/field representation:



Binary/field representation:



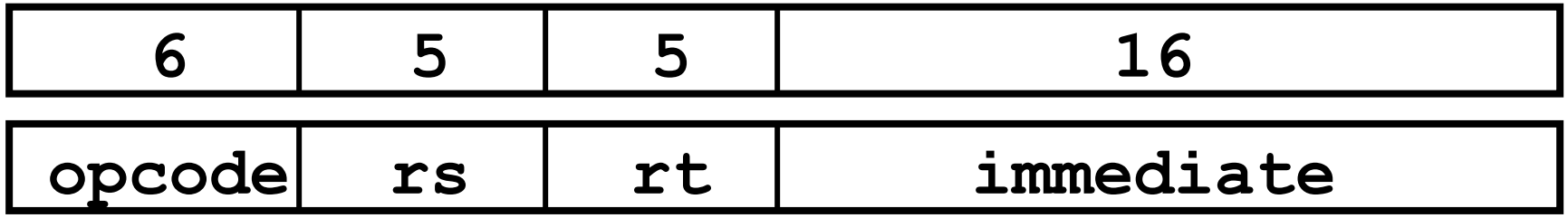
hex representation: $012A4020_{hex}$

- Called a Machine Language Instruction

I-Format Instructions (1/5)

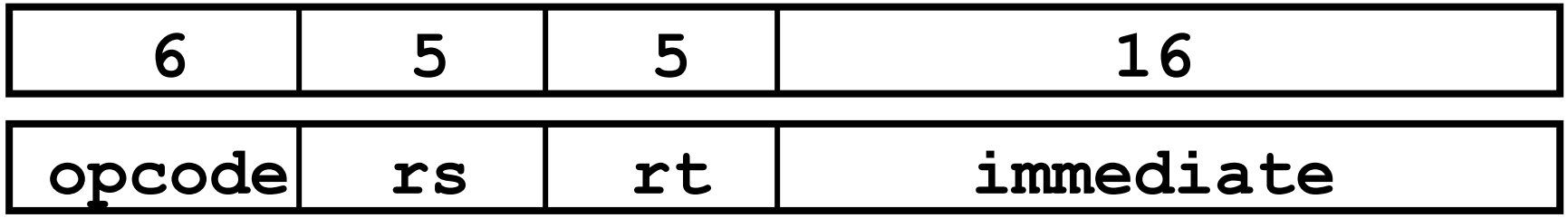
- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31
  - Immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity)
  - Unfortunately, we need to compromise
- Define new instruction format partially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

I-Format Instructions (2/5)



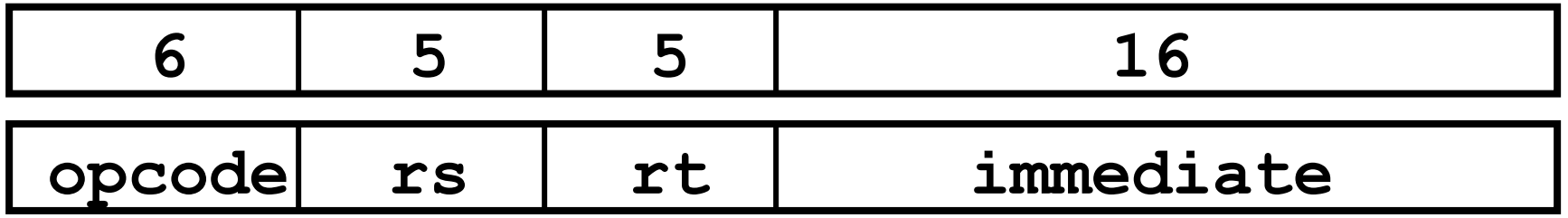
- Define "fields" of a fixed number of bits each
  6 + 5 + 5 + 16 = 32 bits
- Again, each field has a name
- **Key Concept**: Only one field is inconsistent with R-format. Most importantly, ***opcode*** is still in same location.

I-Format Instructions (3/5)



- What do these fields mean?
  - **`opcode`**: same as before, but with no **`funct`** field, **`opcode`** uniquely specifies an instruction in I-format
- ***This finally answers the 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 with other formats

I-Format Instructions (4/5)

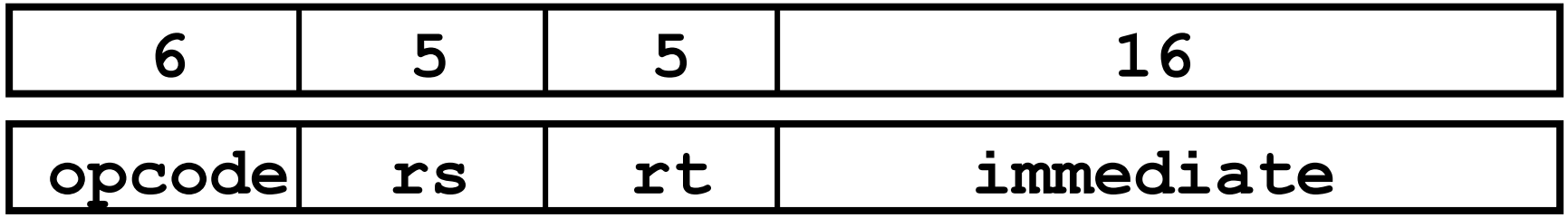


- More fields:

  `rs` specifies the *only* register operand (if there is one)

  `rt` specifies the register which will receive result of the computation (this is why it's called the *target* register "rt")

I-Format Instructions (5/5)



- 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.

I-Format Example (1/2)

- MIPS Instruction:

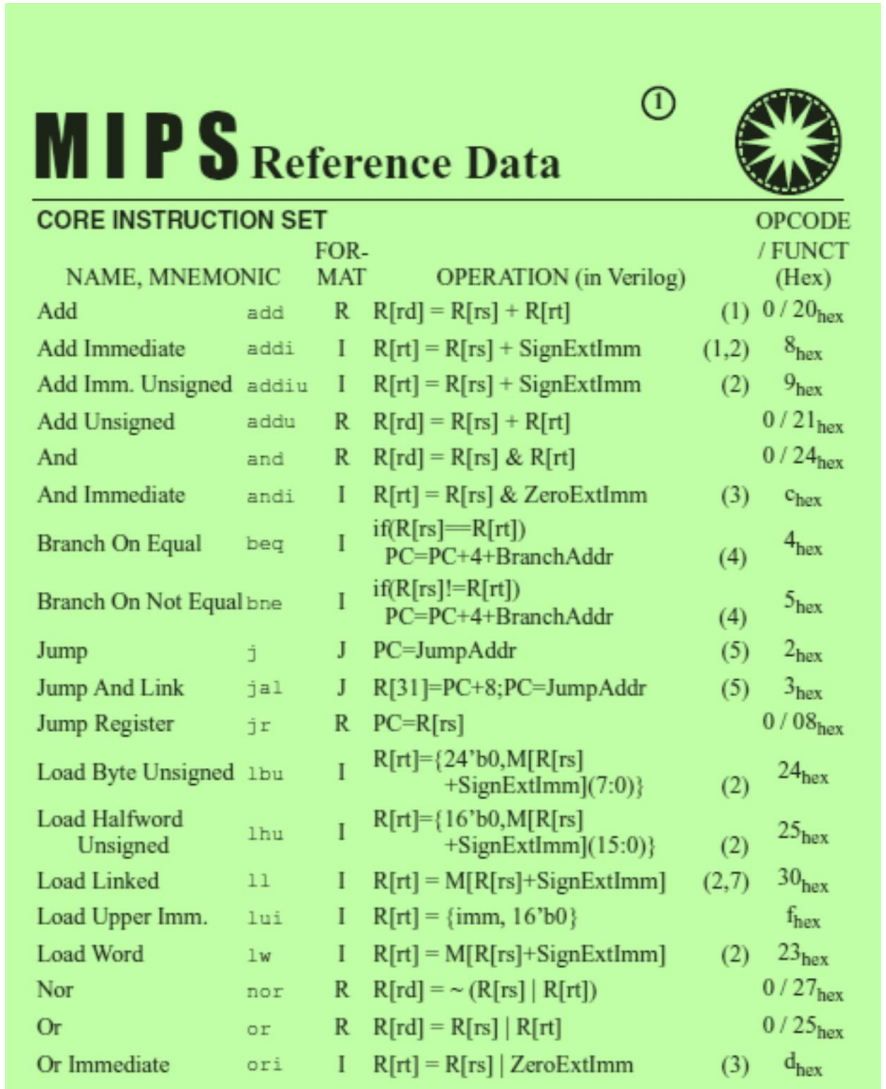
```
addi    $21 $22 -50
```

opcode = 8
(look up in table)

rs = 22
(register containing operand)

rt = 21
(target register)

immediate = -50
(by default, specified in decimal)



MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	or	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}

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: $22D5FFCE_{hex}$

I-Format Problem (1/3)

Problem:

- Chances are that **`addi`**, **`lw`**, **`sw`** and **`slti`** will often use immediates small enough to fit in the immediate field
- But what if the value is too big?
  - We need a way to deal with a 32-bit immediate in a*ny* I-format instruction!

I-Format Problems (2/3)

- Solution to Problem:
  - Handle it in software + new instruction
  - Don’t change the current instructions:
    - Instead, add a new instruction to help out
- New instruction:

```
lui    register immediate
```

  - Stands for Load Upper Immediate
  - Takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register
  - Sets lower half word to zero

I-Format Problems (3/3)

- Solution to Problem (continued):
  - Example of how lui helps:

```
addi  $t0 $t0  0xABABCDCD
```

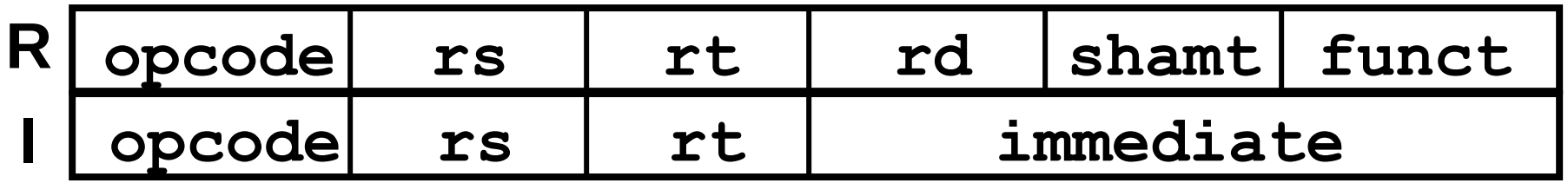
    becomes:

```
lui   $at 0xABAB
ori   $at $at 0xCDCD
add   $t0 $t0 $at
```

  - Now I-format instructions have only 16 bit immediates
  - Assembler can do this for us automatically!
    (more on pseudoinstructions next lecture)

In conclusion

- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (assembler can use **`lw`** and **`sw`**).
- Machine Language Instruction: 32 bits representing a single instruction



- **Remember: The computer actually stores programs as a series of these 32-bit data.**

Review and More Information

- TextBook
  - 2.5 Representing Instructions in the computer
  - 2.10 Addressing for 32-bit immediates
  - 2.12 Translating and Starting a Program
    - Just the section on the **Assembler** with respect to pseudoinstructions (pg 124, 125, 5th edition)