CS 61C: Great Ideas in Computer Architecture
Intro to Assembly Language, MIPS Intro

Instructors:
Krste Asanovic & Vladimir Stojanovic
http://inst.eecs.Berkeley.edu/~cs61c/sp15
Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Compiler

Assembler

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

lw $t0, 0($2)

lw $t1, 4($2)

sw $t1, 0($2)

sw $t0, 4($2)

Anything can be represented as a number, i.e., data or instructions.

0000 1001 1100 0110 1010 1111 0101 1000

1010 1111 0101 1000 0000 1001 1100 0110

1100 0110 1010 1111 0101 1000 0000 1001

0101 1000 0000 1001 1100 0110 1010 1111
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
Using the Stack (1/2)

• So we have a register $sp$ which always points to the last used space in the stack.
• To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
• So, how do we compile this?

```c
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```
Using the Stack (2/2)

- **Hand-compile**

  ```
  int sumSquare(int x, int y) {
  sumSquare:  
    return mult(x,x)+ y; 
  }
  ```

  ```
  addi $sp,$sp,-8 # space on stack
  sw $ra, 4($sp) # save ret addr
  sw $a1, 0($sp) # save y
  add $a1,$a0,$zero # mult(x,x)
  jal mult # call mult
  lw $a1, 0($sp) # restore y
  add $v0,$v0,$a1 # mult()+y
  lw $ra, 4($sp) # get ret addr
  addi $sp,$sp,8 # restore stack
  jr $ra
  ```

  ```
mult: ...```

“push”

“pop”
Basic Structure of a Function

**Prologue**

entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)  # save $ra
save other regs if need be

**Body**  ... (call other functions...)

**Epilogue**

restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
Where is the Stack in Memory?

• MIPS convention
• Stack starts in high memory and grows down
  – Hexadecimal (base 16) : 7fff fffc\text{hex}
• MIPS programs (text segment) in low end
  – 0040 0000\text{hex}
• static data segment (constants and other static variables) above text for static variables
  – MIPS convention global pointer ($\text{gp}$) points to static
• Heap above static for data structures that grow and shrink ; grows up to high addresses
MIPS Memory Allocation

$sp \rightarrow 7fff \ ffff \ c_{hex}$

$gp \rightarrow 1000 \ 8000_{hex}$
$1000 \ 0000_{hex}$

$pc \rightarrow 0040 \ 0000_{hex}$
$0$

Stack

Dynamic data

Static data

Text

Reserved
## Register Allocation and Numbering

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
<th>Preserved on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>The constant value 0</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0–$v1</td>
<td>2–3</td>
<td>Values for results and expression evaluation</td>
<td>no</td>
</tr>
<tr>
<td>$a0–$a3</td>
<td>4–7</td>
<td>Arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0–$t7</td>
<td>8–15</td>
<td>Temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0–$s7</td>
<td>16–23</td>
<td>Saved</td>
<td>yes</td>
</tr>
<tr>
<td>$t8–$t9</td>
<td>24–25</td>
<td>More temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Administrivia

• HW2 due Sunday 2/15
  – Can resubmit HW1’ given 80% weight, and clobber HW1 score
  – Autograder results only available after HW2 deadline
In the News: 1,000 year life Blu-Ray Discs

- Verbatim and Milleniata launch MDISC 100GB Blu-ray optical media with >1,000 year lifetime
- “average lifetime of significantly more than 2,000 years with no more than one failure per 100,000 discs expected, after more than 1,000 years at 250C (770F) and 50 per cent relative humidity.”
Levels of Representation/Interpretation

High Level Language Program (e.g., C) → Compiler → Assembly Language Program (e.g., MIPS) → Assembler → Machine Language Program (MIPS)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Register File

ALU

0000 1001 1100 0110 1010 1111 0101 1000 1010 1111 0101 1000 0000 1001 1100 0110 1100 0110 1010 1111 0101 1000 0000 1001 0101 1000 0000 1001 1100 0110 1010 1111

Anything can be represented as a number, i.e., data or instructions

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)

Anything can be represented as a number, i.e., data or instructions.
Big Idea:
Stored-Program Computer

– 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
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 (Pentium 4); 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.
  – \texttt{lw} and \texttt{sw} both access memory one word at a time.

• So how do we represent instructions?
  – Remember: Computer only understands 1s and 0s, so “\texttt{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. E.g.,
    • `mult` and `div` have nothing important in the `rd` field since the dest 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)
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:
  \[ \text{add} \quad \$8, \$9, \$10 \]

  - \text{opcode} = 0 (look up in table in book)
  - \text{funct} = 32 (look up in table in book)
  - \text{rd} = 8 (destination)
  - \text{rs} = 9 (first \textit{operand})
  - \text{rt} = 10 (second \textit{operand})
  - \text{shamt} = 0 (not a shift)
R-Format Example (2/2)

• MIPS Instruction:
  \texttt{add \$8,\$9,\$10}

Decimal number per field representation:

\begin{array}{ccccccc}
0 & 9 & 10 & 8 & 0 & 32 \\
\end{array}

Binary number per field representation:

\begin{array}{ccccccc}
000000 & 01001 & 01010 & 01000 & 00000 & 100000 \\
\end{array}

hex representation: \quad 012A \ 4020_{\text{hex}}

decimal representation: \quad 19,546,144_{\text{ten}}

Called a \textbf{Machine Language Instruction}
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:
  \[6 + 5 + 5 + 16 = 32 \text{ bits}\]

\[
\begin{array}{cccc}
6 & 5 & 5 & 16 \\
\end{array}
\]

– Again, each field has a name:

\[
\begin{array}{cccc}
\text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
\end{array}
\]

– Key Concept: Only one field is inconsistent with R-format. Most importantly, \textbf{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:
  \texttt{addi} \quad \$21,\$22,-50

\texttt{opcode} = 8 (look up in table in book)
\texttt{rs} = 22 (register containing operand)
\texttt{rt} = 21 (target register)
\texttt{immediate} = -50 (by default, this is decimal in assembly code)
I-Format Example (2/2)

- MIPS Instruction:
  \[
  \text{addi} \quad \$21,\$22,-50
  \]

### Decimal/field representation:

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>22</th>
<th>21</th>
<th>-50</th>
</tr>
</thead>
</table>

### Binary/field representation:

|   | 001000 | 10110 | 10101 | 11111111111001110 |

### Hexadecimal representation:

- \( 22D5 \) \(_{\text{hex}}\)

### Decimal representation:

- \( 584,449,998 \) \(_{\text{ten}}\)
Clicker/Peer Instruction

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Binary Representation</th>
</tr>
</thead>
</table>
| a) add $0, 0, 0$ | \begin{array}{cccccc} \text{opcode} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
| b) subu $s0, s0, s0$ | \begin{array}{cccccc} \text{opcode} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
| c) lw $0, 0(0)$ | \begin{array}{cccccc} \text{opcode} & \text{rs} & \text{rt} & \text{offset} \\
| d) addi $0, 0, 35$ | \begin{array}{cccccc} \text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
| e) subu $0, 0, 0$ | \begin{array}{cccccc} \text{opcode} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\

Registers numbers and names:
- $0$: $0$,
- $8$: $t0$,
- $9$: $t1$,
- $15$: $t7$,
- $16$: $s0$,
- $17$: $s1$,
- $23$: $s7$

Opcodes and function fields (if necessary):

- **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`CD
  – This is a pseudo-instruction!

• Translates into:

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

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

  Only the assembler gets to use $at
Branching Instructions

• `beq` and `bne`
  – Need to specify a target address if branch taken
  – Also specify two registers to compare

• Use I-Format:

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| opcode | rs | rt | immediate |
```

  – 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

• So just how much of memory can we reach?
Branching Reach

- **Recall:** MIPS uses 32-bit addresses
  - Memory is byte-addressed

- **Instructions are** *word-aligned*
  - Address is always multiple of 4 (in bytes), meaning it ends with 0b000 in binary
  - Number of bytes to add to the PC will always be a multiple of 4

- **Immediate specifies words instead of bytes**
  - Can now branch $\pm 2^{15}$ words
  - We can reach $2^{16}$ instructions = $2^{18}$ bytes around PC
Branch Calculation

- **If we don’t take the branch:**
  - $PC = PC + 4 = \text{next instruction}$

- **If we do take the branch:**
  - $PC = (PC + 4) + (\text{immediate} \times 4)$

- **Observations:**
  - $\text{immediate}$ 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  
    rs = 9  
    rt = 0  
    immediate = 3

(look up on Green Sheet)
(first operand)
(second operand)

Start counting from instruction AFTER the branch
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):

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|  4  |  9 |  0 |  3 |
```

Field representation (binary):

```
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 |  9 |  8 |  7 |  6 |  5 |  4 |  3 |  2 |  1 |  0 |
|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 000100 | 01001 | 00000 | 00000000000000000011 |
```
Questions on PC-addressing

• Does the value in branch immediate field change if we move the code?
  – If moving individual lines of code, then yes
  – If moving all of code, then no

• What do we do if destination is $> 2^{15}$ instructions away from branch?
  – Other instructions save us

\[
\begin{align*}
\text{beq} & \ $s0,\$0, \text{far} & \text{bne} & \ $s0,\$0, \text{next} \\
\# \text{ next instr} & \rightarrow & \text{j far} & \text{next:} \ # \text{ next instr}
\end{align*}
\]
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 jal), 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:

```
31 6 26 0
```

• As usual, each field has a name:

```
31

opcode target address
```

• 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 0b00 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:

  \texttt{move \, dst, src} \texttt{translated into}

  \texttt{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 \( \text{dst} \)

• **Load Immediate (li)**
  – li dst,imm
  – Loads 32-bit immediate into \( \text{dst} \)

• **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
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

• TAL ⊆ MAL
Summary

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

  **I:**
  
  \[
  \begin{array}{|c|c|c|c|}
  \hline
  \text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
  \hline
  \end{array}
  \]

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

  **J:**
  
  \[
  \begin{array}{|c|}
  \hline
  \text{opcode} & \text{target address} \\
  \hline
  \end{array}
  \]

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

  **R:**
  
  \[
  \begin{array}{|c|c|c|c|c|c|}
  \hline
  \text{opcode} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  \hline
  \end{array}
  \]