COMPUTER ORGANIZATION (IS F242)

LECT 23: MIPS ARCHITECTURE

MIPS Architecture

Microprocessor without Interlocked Pipeline Stages

Registers and their Uses in MIPS

$zero	0	The constant value 0
$at	1	Assembler Temporary
$v0 - $v1	2-3	Values for function results and Expression evaluation
$a0 - $a3	4-7	Arguments
$t0 - $t7	8-15	Temporaries
$s0 - $s7	16-23	Saved temporaries
$t8 - $t9	24-25	Temporaries
$k0 - $k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Registers in MIPS

Preserved	Not Preserved
Saved Registers: $s0 - $s7	Temporary registers: $t0 - $t9
Stack pointer register: $sp	Argument registers: $a0 - $a3
Return address register: $ra	Return value registers: $v0, $v1
Stack above the stack pointer	Stack below the stack pointer

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Examples- C Programs in MIPS

f = (g + h) – (i + j);

$s0, $s1, $s2, $s3, $s4

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example 1

- C code:

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

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Examples- C Programs in MIPS

Q1	A[300] = h + A[100]; A → $s1, h → $s2
Q2	A[100] = A[100] + A[100]; A → $s3
Q3	A[200] = A[100] + A[100]; A[100] = A[200] + A[100]; A → $s4

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 4
```

- No subtract immediate instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

```
add $t2, $s1, $zero
```

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

Addressing Modes in MIPS – R Type

- Example:

```
add $t0, $s1, $s2
sub $s0, $t0, $t1
and $t0, $t1, $t2
```

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

R-format Example

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

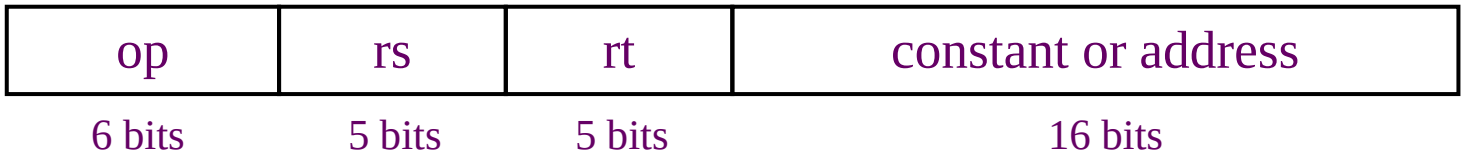
$00000010001100100100000000100000_2 = 02324020_{16}$

Addressing Modes in MIPS – I Type

- I Type
  - Examples:

```
lw  $t0, 32($s3)
sw  $t0, 48($s3)
addi $s3, $s3, 4
beq rs, rt, L1
bne rs, rt, L1
```

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Addressing Modes in MIPS – J Type

- J Type
  - `Examples: j L1`

Instruction Class

- 5 Classes
  - Integer arithmetic and Logic
  - Data movement
  - Integer multiply / Divide
  - Flow control (branch)
  - Floating point arithmetic
- Plus
  - System control instructions for use of OS

Signed / Unsigned

- Signed numbers are 2's complement
- Sign bit 0 / 1
- Overflow Vs Carry
- Sign Vs Zero extensions
- Exceptions

Arithmetic Instructions

- add
  - add rd, rs, rt
  - add $s1, $s2, $s3 # $s1 = $s2 + $s3
  - Exception possible
- add unsigned
  - addu rd, rs, rt
  - addu $s1, $s2, $s3 #$s1 = $s2 + $s3
  - No Exceptions

Arithmetic Instructions

- sub
  - sub rd, rs, rt
  - sub $s1, $s2, $s3 # $s1 = $s2 - $s3
  - Exception possible
- sub unsigned
  - subu rd, rs, rt
  - subu $s1, $s2, $s3 #$s1 = $s2 - $s3
  - No Exceptions

Multiply

- Hi and lo registers
- Signed / Unsigned
- mult
  - mult rs, rt
  - mult $s2, $s3 # hi, lo = $s2 * $s3 signed product
- multu
  - multu rs, rt
  - multu $s2, $s3 # hi, lo = $s2 * $s3 unsigned product