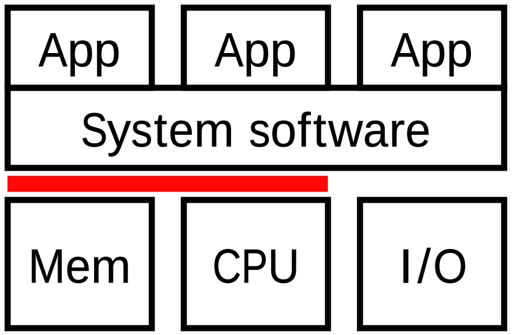
Computer Architecture

Lec 3: ISA

Review: Program Compilation



```
int array[100], sum;
void array_sum() {
   for (int i=0; i<100;i++) {
      sum += array[i];
   }
}
```

- **Program** written in a “high-level” programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
  - Compiler itself another application ... who compiled compiler?

Review: Instructions (insn)

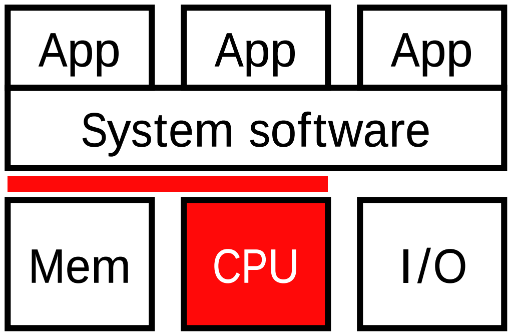
```
int array[100], sum;
void array_sum() {
   for (int i=0; i<100;i++)
   {
       sum += array[i];
   }
}
```

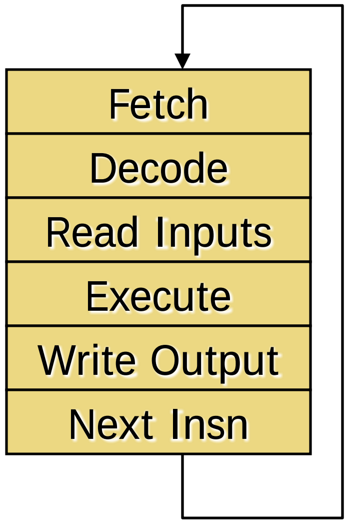
```
    .data
array: .space 100
sum:   .word 0
    .text
array_sum:
    li $5, 0
    la $1, array
    la $2, sum
as_loop:
    lw $3, 0($1)
    lw $4, 0($2)
    add $4, $3, $4
    sw $4, 0($2)
    addi $1, $1, 1
    addi $5, $5, 1
    li $6, 100
    blt $5, $6, as_loop
```

Instructions: Bit-patterns hardware interprets as commands

Assembly Language: insn designed to be readable by humans

Review: Insn Execution Model





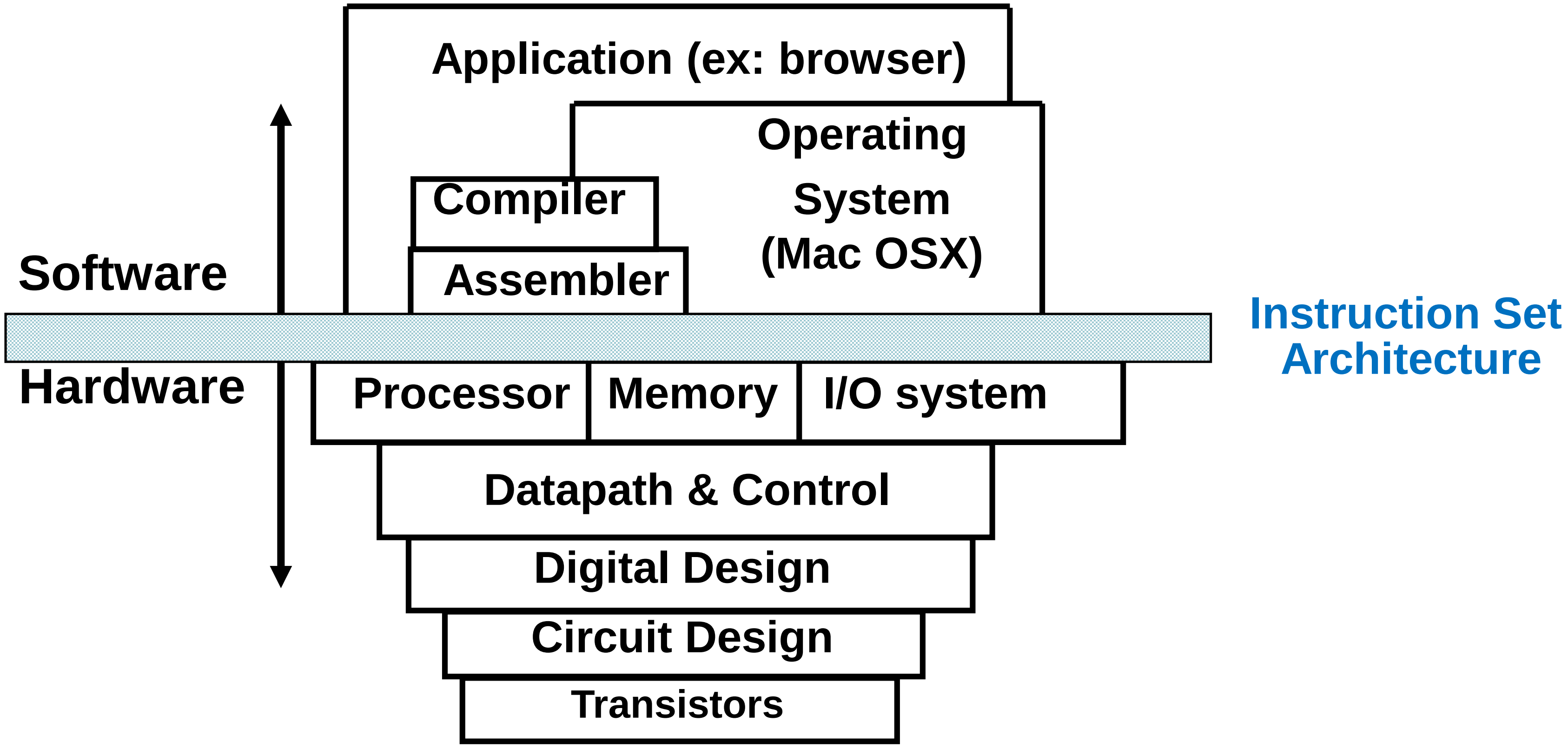
Instruction → Insn

- The computer is just finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Sometimes called "instruction pointer"
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just "data in memory"**
  - Makes computers programmable ("universal")

What Is An ISA?

- **ISA (instruction set architecture)**
  - A well-defined hardware/software interface
  - The **“contract”** between software and hardware
    - **Functional definition** of storage locations & operations
      - Storage locations: registers, memory
      - Operations: add, multiply, branch, load, store, etc
    - **Precise description** of how to invoke & access them
- Not in the “contract”: non-functional aspects
  - How operations are implemented
  - Which operations are fast and which are slow
  - Which operations take more power and which take less

What Is An ISA?



Some Key Attributes of ISAs

- Instruction encoding
  - Fixed length (32-bit for MIPS)
  - Variable length (x86 1 byte to 16 bytes, average of ~3 bytes)
  - Limited variability (ARM insns are 16b or 32b)
- Number and type of registers
  - MIPS has 32 “integer” registers and 32 “floating point” registers
  - ARM & x86 both have 16 “integer” regs and 16 “floating point” regs
- Address space
  - ARM: 32-bit addresses at 8-bit granularly (4GB total)
  - Modern x86 and ARM64: 64-bit addresses (16 exabytes!)
- Memory addressing modes
  - MIPS: address calculated by “reg+offset”
  - x86 and others have much more complicated addressing modes

CISC vs. RISC

- Complex Instruction Set Computer (CISC)
  - A single instruction can be used to do all of the loading, evaluating and storing operations
  - Minimise the number of instructions per programme
  - Increases the number of cycles per instruction
  - E.g., x86
- Reduced Instruction Set Computer (RISC)
  - Instruction set is composed of a few basic steps for loading, evaluating and storing operations
  - Reduces the number of cycles per instruction
  - Increase the number of instructions per program
  - E.g., MIPS, ARM, RISC-V etc

CISC vs. RISC

- RISC
  - Emphasis on software
  - Small number of fixed length instructions
  - Simple, standardised instructions
  - Single clock cycle instructions
  - Low cycles per insn with large code sizes

- CISC
  - Emphasis on hardware
  - Large number of instructions
  - Complex, variable-length instructions
  - Instructions can take several clock cycles
  - Small code sizes with high cycles per insn

RISC vs. CISC: Which is better? (x86 is CISC, but transfers each insn to several micro-insns…)

Popular ISAs

- Intel x86 (laptops, desktop, and servers)
- MIPS (used throughout in book)
- ARM (in all your mobile devices)
- PowerPC (servers & game consoles)
- SPARC (servers)
- RISC-V (an open source ISA)
- Historical: IBM 370, VAX, Alpha, PA-RISC, 68k, ...

A RISC Example - MIPS

- MIPS basic instructions
  - Arithmetic instructions: add, addi, sub
  - Data transfer instructions: lw, sw, lb, sb
  - Control instructions: bne, beq, j, slt, slti
  - Logical operations: and, andi, or, ori, nor, sll, srl
- MIPS instruction format
  - R-format, I-format, J-format
- Encoding/decoding assembly code
  - Disassembly starts with opcode
  - Pseduo instructions are introduced

MIPS Registers

- 32 registers in MIPS
  - Why 32? Design principle: Smaller is faster
  - Registers are numbered from 0 to 31
- Each register can be referred to by number or name
  - Number references: **`$0, $1, … $30, $31`**
  - By convention, each register also has a name to make it easier to code
    - **`$t0 - $t7`** for temporary variables ($8- $15)
    - **`$ra`** for return address
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

MIPS Register Convention

Name	**Register Number**	**Usage**	**Preserve on call?**
$zero	0	constant 0 (**hardware**)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	**yes**
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	**yes**
$t8 - $t9	24-25	temporaries	no
$k	26-27	Interrupt/trap handler	**yes**
$gp	28	global pointer	**yes**
$sp	29	stack pointer	**yes**
$fp	30	frame pointer	**yes**
$ra	31	return addr (**hardware**)	**yes**

MIPS Instructions

- Syntax of instructions:

  `op    dest, src1, src2`

  - Op: operation by name
  - Dest: operand getting result (“destination”)
  - Src1: 1st operand for operation (“source1”)
  - Src2: 2nd operand for operation (“source2”)
- Each line of assembly code contains at most 1 instruction
- Hash (#) is used for MIPS comments
  - Anything from hash mark to end of line is a comment and will be ignored
  - Every line of your comments must start with a #

Addition/Subtraction Example

- How to do the following C statement?

  `a = b + c + d - e;`

- Break into multiple instructions
  - `add $t0, $s1, $s2 #temp = b + c`
  - `add $t0, $t0, $s3 #temp = temp + d`
  - `sub $s0, $t0, $s4 #a = temp - e`
- Notice
  - A single line of C code may break up into several lines of MIPS code
  - May need to use temporary registers ($t0 - $t9) for intermediate results

Constant or Immediate Operands

- Immediates are numerical constants
  - They appear often in code, so there are special instructions for them
  - Design principle: Make the common case fast
- Add Immediate:
  - C code : f = g + 10
  - MIPS code: addi $s0,$s1,10
    - MIPS registers $s0, $s1 are associated with C variables f, g
  - Syntax similar to add instruction, except that last argument is a number instead of a register
  - How about subtraction? subi?

Constant or Immediate Operands

- There is NO subtract immediate instruction in MIPS: Why?
  - ISA design principle: limit types of operations that can be done to minimum
  - If an operation can be decomposed into a simpler operation, do not include it
  - addi ..., -X = subi ..., X => so no subi
- Example
  - C code: f = g - 10
  - MIPS code: addi $s0,$s1,-10
    - MIPS registers $s0,$s1 are associated with C variables f, g

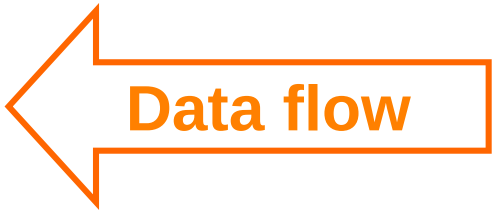
Data Transfer Instructions

- MIPS has two basic data transfer instructions for accessing memory

```
lw $t0,4($s3)  #load word from memory
sw $t0,8($s3)  #store word to memory
```

- Load instruction syntax: lw reg1, offset(reg2)
  - Operator name: lw (meaning Load Word, so 32 bits or one word are loaded at a time)
  - Reg1: register that will receive the transferred data
  - Offset: a numerical offset in bytes
  - Reg2: register containing pointer to memory, called base register

Load Word Example



- Example: lw $t0,12($s0)
  - This instruction will take the pointer in $s0, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register $t0
    - $s0 is called the base register
    - 12 is called the offset
  - Offset is generally used in accessing elements of array or structure: base register points to beginning of array or structure

Store Instruction

- Also want to store from register into memory
  - sw: meaning Store Word, so 32 bits or one word are loaded at a time)
  - Store instruction syntax is identical to Load's

- Example: sw $t0,12($s0)
  - This instruction will take the pointer in $s0, add 12 bytes to it, and then store the value from register $t0 into that memory address
  - Remember: "Store INTO memory"

MIPS Decision Instructions

- Decision instructions in MIPS
  - **`beq register1, register2, L1`**
    - beq is “branch if equal”
    - same meaning as: if (register1==register2) goto L1
  - **`bne register1, register2, L1`**
    - bne is “branch if not equal”
    - same meaning as: if (register1!=register2) goto L1

- Called *conditional branches*
  - Can be used to implement complex control-flow constructs for high level langauages

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an *unconditional branch*:

```
j        label
```

  - Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition
  - Same meaning as: goto label
- Technically, it's the same as:
  - beq $0,$0,label
    - Condition always satisfied

Inequalities in MIPS (1/2)

- General programs need to test < and > as well as equalities (== and != in C)
- MIPS inequality instruction:

  ```
  slt reg1,reg2,reg3
  ```

  - "Set on Less Than"
  - Meaning: reg1 = (reg2 < reg3);
    - if (reg2 < reg3) reg1 = 1;
    - else reg1 = 0;
  - In computereeze, "set" means "set to 1", "reset" means "set to 0"

Inequalities in MIPS (2/2)

- Now, we can implement <, but how do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
    - MIPS goal: simpler is better
- Can we implement ≤ in one or more instructions using just slt and the branches?
- What about >?
- What about ≥?

Logical Operators

- Logical instruction syntax:

  ```
  op   dest, src1, src2
  ```

  - Op: operation name (**`and, or, nor`**)
  - Dest: register that will receive value
  - Src1: first operand (register)
  - Src2: second operand (register) or immediate
- Accept exactly 2 inputs and produce 1 output
  - Benefit: rigid syntax$\Rightarrow$ simpler hardware
- Immediate operands
  - **`andi, ori`**: both expect the third argument to be an immediate

Uses for Logical Operators (1/3)

- Use **AND** to create a mask
  - Anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit
- Example:

1011 0110 1010 0100 0011 1101 1001 1010

0000 0000 0000 0000 0000 1111 1111 1111

0000 0000 0000 0000 0000 1101 1001 1010

Mask retaining the last 12 bits

Uses for Logical Operators (2/3)

- A bit pattern in conjunction with **AND** is called a mask that can conceal some bits
  - The previous example a mask is used to isolate the rightmost 12 bits of the bit-string by masking out the rest of the string (e.g. setting it to all 0s)
  - Concealed bits are set 0s, while the rest bits are left alone
  - In particular, if the first bit-string in the above example were in $t0, then the following instruction would mask it:

```
andi   $t0, $t0, 0xFFF
```

Uses for Logical Operators (3/3)

- Similarly effect of **OR** operation
  - Oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit
  - This can be used to force certain bits to 1s
- Example
  - $t0 contains 0x12345678, then after this instruction:

```
ori  $t0, $t0, 0xFFFF
```

  - $t0 contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s)

Logical Shift Instructions

- Shift instruction syntax:

  op dest,reg,amt

  - Op: operation name
  - Dest: register that will receive value
  - Reg: register with the value to be shifted
  - Amt: shift amount (constant < 32)
- MIPS logical shift instructions:
  - **sll** (shift left logical): shifts left and fills emptied bits with 0s
  - **srl** (shift right logical): shifts right and fills emptied bits with 0s
  - MIPS also has arithmetic shift instructions that fills with the sign bit

Instruction Representation

- Instructions in MIPS are 32-bit long (one word) and divided into “fields”
  - Each field tells computer something about an instruction
- We could define different fields for each instruction, but MIPS defines only three basic types of instruction formats due to simplicity
  - R-format: register format
  - I-format: immediate format
  - J-format: jump format

Instruction Formats

- I-format: immediate format
  - Instructions with immediates
    - Excluding shift instructions
  - Data transfer instructions (since the offset counts as an immediate)
  - Branches (beq and bne)
- J-format: jump format
  - j and jal (more details later)
- R-format: used for all other instructions
- It will soon become clear why the instructions have been partitioned in this way

R-Format Instructions (1/4)

- Define six fields of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

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

  - Each field has a name
  - 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 (00000 - 11111) while 6-bit fields can represent any number 0-63 (000000-111111)

R-Format Instructions (2/4)

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

- opcode: partially specifies the operation
  - Also implies the instruction format: opcode=0 for all R-type instructions
- funct: combined with opcode, exactly specifies the instruction
- rs (source register): *generally* register containing the 1st operand
- rt (target register): *generally* register containing the 2nd operand (note that name is misleading)
- rd (destination register): *generally* register which will receive the result of computation

R-Format Instructions (3/4)

- 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 “generally” was used because there are exceptions that we’ll see later
    - E.g. multiplication will generate a result of 64 bit stored in two special registers: nothing important in the rd field

R-Format Instructions (4/4)

- Final field: shamt
  - Shift amount: 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

R-Format Example

- MIPS Instruction: add $8,$9,$10
  - Encode to decide the value of each field
    - opcode = 0, funct = 32 (look up in table in book)
    - rd = 8 (destination)
    - rs = 9 (first operand), rt = 10 (second operand)
    - shamt = 0 (not a shift)
  - Decimal number per field representation

0	9	10	8	0	32

  - Binary number per field representation

000000	01001	01010	01000	00000	100000

hex

  - Machine language instruction:
    - Hex representation: 012A 4020$_{hex}$
    - Decimal representation: 19,546,144$_{ten}$

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
  - Ideally, MIPS would have only one instruction format for simplicity: unfortunately, we need to compromise

  - Still, try to define new instruction format that is partially consistent with R-format
    - The first three fields of both formats are the same size and have the same names
    - The rest three fields in R-format are merged to form a single field for the immediate operand

I-Format Instructions (2/4)

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

6	5	5	16
`opcode`	`rs`	`rt`	`immediate`

  - Again, each field has a name
  - Design key
    - Only one field is inconsistent with R-format
    - Most importantly, opcode is still in the same location

I-Format Instructions (3/4)

6	5	5	16
`opcode`	`rs`	`rt`	`immediate`

- opcode: uniquely specifies an instruction
  - No funct field
- rs: specifies the only 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”

I-Format Instructions (4/4)

- The immediate field
  - Used to specify immediates for instructions with a numerical constant operands
  - Used to specify address offset in data transfer instructions: lw, sw, etc.
  - Used to specify branch address in bne and beq

  - Range
    - Both positive and negative numbers
    - 16 bits ➔ can be used to represent immediate up to 2^{16} different values
    - What if the number we want to represent is out of the range?

I-Format Example

- MIPS Instruction: addi $21,$22,-50
  - Encode for each field
    - opcode = 8 (look up in table in book)
    - rs = 22 (register containing operand)
    - rt = 21 (target register)
    - immediate = -50 (by default, this is decimal)

Negative number encoding: 2's complement

  - Decimal number per field representation

8	22	21	-50

  - Binary number per field representation

001000	10110	10101	1111111111001110

  - Hexadecimal representation: 22D5 FFCE$_{\text{hex}}$
    Decimal representation: 584,449,998$_{\text{ten}}$

Large Immediates

- Range of immediates is limited
  - Length of immediate field is 16 bits
  - Considered as a signed number (sign bit)
- Arithmetic operands or address offset can be larger
  - 32-bit data / address in MIPS
  - We need a way to deal with a 32-bit immediate in any I-format instruction
- Solution:
  - Handle it in software + new instruction
  - Don’t change the current instructions: instead, add a new instruction to help out

Large Immediates

- New instruction:

  lui register, immediate

  - Load Upper Immediate
  - Takes 16-bit immediate and puts these bits in the upper half (high order half) of the specified register; lower half is set to 0s
  - Example:
    - Want to write: **`addi $t0,$t0, 0xABABCDCD`**
    - Need to write a sequence instead:

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

Immediates in Conditional Branches

- Branch instructions bne and beq

opcode	rs	rt	immediate

  - Field rs and rt specify registers to compare
  - Field immediate specify branch address
    - 16 bit is too small since we have 32-bit pointer to memory
- Observation
  - Branches are used for if-else, while-loop, for-loop: tend to branch to a nearby instruction
  - We only need to know the difference between the branch target and the current instruction address, which is much smaller and 16-bit addressing might suffice in most cases

PC-Relative Addressing

- Solution to branches in a 32-bit instruction: PC-relative addressing
  - PC is the special register containing the address of the current instruction
  - New program counter = PC + branch address
    - Let the 16-bit immediate field be a signed two’s complement integer to be added to the PC if we take the branch
- Now we can branch $\pm 2^{15}$ bytes from the PC, which should be enough to cover almost any loop
  - Any ideas to further optimize this?

PC-Relative Addressing

- Note: Instructions are words, so they are word aligned
  - The byte address of an instruction is always a multiple of 4, i.e. it must end with 00 in binary
  - $\Rightarrow$The number of bytes to add to the PC will always be a multiple of 4
  - $\Rightarrow$Specify the immediate in words
- Now, we can branch $\pm$ 2^{15} words from the PC (or $\pm$ 2^{17} bytes), so we can handle loops 4 times as large

Branch Address Calculation

- Calculation:
  - If we do not take the branch:

    PC = PC + 4

    - PC+4 = byte address of next instruction
  - If we do take the branch:

    PC = (PC + 4) + (immediate * 4)
- Observations
  - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump
  - Immediate field can be positive or negative
  - Due to hardware, add immediate to (PC+4), not to PC; will be clearer why later in course

Branch Example

- MIPS Code:

  -
    ```
    Loop: beq   $9,$0,End
          add   $8,$8,$10
          addi  $9,$9,-1
          j     Loop
    End:
    ```
- Encoding in I-Format:
  - opcode = 4 (look up in table)
  - rs = 9 (first operand)
  - rt = 0 (second operand)
  - immediate field: no. of instructions to add to (or subtract from) the PC, starting at the instruction following the branch
    - Here, immediate = 3

Branch Example

- MIPS Code:
  -
```
Loop: beq   $9, $0, End
      add   $8, $8, $10
      addi  $9, $9, -1
      j     Loop
End:
```
- Decimal representation

4	9	0	3

- Binary representation

000100	01001	00000	0000000000000011

J-Format Instructions

- J-format is used by MIPS jump instructions
  - j and jal
  - 6-bit opcode + 26-bit jump address

6 bits	26 bits
opcode	target address

- Key concepts
  - Keep opcode field identical to R-format and I-format for consistency
  - Combine all other fields to make room for large target address
    - Goto statements and function calls tend to have larger offsets than branches and loops

J-Format Addressing

- We have 26 bit to specify the target address
  - We cannot fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise
  - Like branches, jumps will only jump to word aligned addresses $\Rightarrow$ the 26-bit field covers 28 bits of the 32-bit address space
- Where do we get the other 4 bits?
  - Take the 4 highest order bits from the PC
  - Technically, this means that we cannot jump to anywhere in memory, but it's adequate 99.9999...% of the time, since programs aren't that long
    - Only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the **`jr`** instruction

MIPS Addressing Modes

- Register addressing (R-Type)
  - Operand is stored in a register
- Base or displacement addressing (I-Type)
  - Operand at the memory location specified by a register value plus a displacement given in the instruction; Eg: lw, $t0, 25($s0)
- Immediate addressing (I-Type)
  - Operand is a constant within the instruction itself
- PC-relative addressing (I-Type)
  - The address is the sum of the PC and a constant in the instruction
- Pseudo-direct addressing (J-type)
  - New PC = {(upper 4 bits of PC+4), 26-bit constant, 00}

Decoding Machine Language

- How do we convert 1s and 0s to C code?
  - Machine language $\Rightarrow$ Assembly language $\Rightarrow$ C?
- For each 32 bits:
  - Look at opcode: 0 means R-Format, 2 or 3 mean J-Format, otherwise I-Format
  - Use instruction type to determine which fields exist
  - Write out MIPS assembly code, converting each field to name, register number/name, or decimal/hex number
  - Logically convert this MIPS code into valid C code

Decoding Example (1/5)

- Here are six machine language instructions in hexadecimal:

```
00001025hex
0005402Ahex
11000003hex
00441020hex
20A5FFFFhex
08100001hex
```

  - Let the first instruction be at address 4,194,304ten (0x00400000hex)
- Next step: convert hex to binary

Decoding Example (2/5)

- The six machine language instructions in binary:

```
R 00000000000000000001000000100101
R 00000000000001010100000000101010
I 00010001000000000000000000000011
R 00000000010001000001000000100000
I 00100000101001011111111111111111
J 00001000000100000000000000000001
```

- Next step: identify opcode and format

R	0	rs	rt	rd	shamt	funct
I	1, 4-31	rs	rt	immediate		
J	2 or 3	target address				

Decoding Example (3/5)

- Next: fields separated based on format / opcode:

R	0	0	0	2	0	37
R	0	0	5	8	0	42
I	4	8	0	+3		
R	0	2	4	2	0	32
I	8	5	5	-1		
J	2	1,048,577				

- Next step: translate (disassemble) to MIPS instructions

Decoding Example (4/5)

- MIPS assembly (Part 1):

Address	Assembly instructions	
`0x00400000`	`or`	`$2,$0,$0`
`0x00400004`	`slt`	`$8,$0,$5`
`0x00400008`	`beq`	`$8,$0,3`
`0x0040000c`	`add`	`$2,$2,$4`
`0x00400010`	`addi`	`$5,$5,-1`
`0x00400014`	`j`	`0x100001`

- Better solution: translate to more meaningful MIPS instruction (fix the branch/jump, add labels and register names)

Decoding Example (5/5)

- MIPS Assembly (Part 2):

```
          or     $v0,$0,$0
Loop:     slt    $t0,$0,$a1
   beq     $t0,$0,Exit
   add     $v0,$v0,$a0
   addi    $a1,$a1,-1
   j       Loop
Exit:
```

- Next step: translate to C code (be creative!)

```
product = 0;
while (multiplier > 0) {
    product += multiplicand;
    multiplier -= 1;
}
```

$v0: product
$a0: multiplicand
$a1: multiplier

MIPS Instruction Formats Summary

- Minimum number of instructions required
  - Information flow: load/store
  - Logic operations: logic and/or/nor, shift
  - Arithmetic operations: addition, subtraction, etc.
  - Branch operations: bne, beq
  - Jump operations: j, jal
- Instructions have different number of operands
- 32 bits representing a single instruction

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Summary

- MIPS instruction set architecture
  - MIPS basic instructions
    - Arithmetic instructions: add, addi, sub
    - Data transfer instructions: lw, sw, lb, sb
    - Control instructions: bne, beq, j, slt, slti
    - Logical operations: and, andi, or, ori, nor, sll, srl
  - MIPS instruction format
    - R-format, I-format, J-format
  - Encoding/decoding assembly code
    - Disassembly starts with opcode
    - Pseduoinstructions are introduced