Instruction Set Architecture

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- What is instruction set architecture (ISA)?
- Computer operations
- Operands
- Logical operations
- Operations for making decision
- Procedure/Function calls
- Instruction encoding
- Manipulating strings
- Floating-point instructions
- Translating and starting a program

What is ISA?

- Operations of a computer are done through instructions
  - One instruction does one operation
- Set of instructions supported by a computer is the instruction set of the computer
- Organization of the computer defined by the instruction set is the instruction set architecture (ISA) of the computer
- The guiding principle in designing ISA is that the implementation of the ISA should be simple
- The programming language that uses instructions of a computer is called an assembly language

Computer operations

- Every computer must be able to do arithmetic operations
  - Add, subtract, multiply, divide
  - The variables operated on and the result variables are called operands
    - The operands used in an operation are called source operands or read operands
    - The operands used for storing the result of an operation are called destination operands or write operands
    - c = a + b: a and b are source or read operands, c is the destination or write operand
    - Most computers allow at most two sources and one destination per instruction

Computer operations

- Consider the MIPS instruction "add a, b, c"
  - MIPS is the name of a processor family
    - We will understand what MIPS stands for later in the course
  - "add" is the name of the instruction which defines the operation to be done when this instruction is encountered by the computer
  - Among the three operands (a, b, c), the one right after "add" is the destination operand
    - Usually the destination operand is mentioned right after the operation name
  - Other two operands (b, c) are source operands
  - Add b and c, put the result in a

Computer operations

- The MIPS add instruction can be used to generate an instruction sequence for adding a larger number of operands
  - Consider adding b, c, d, e and storing the result in a (C language statement: a = b+c+d+e;)

```
add a, b, c
add a, a, d
add a, a, e
```

- A compiler (e.g., gcc) generates a sequence of instructions from a HLL (e.g., C) program
  - Each instruction specifies one operation
  - The program containing this sequence of MIPS instructions is called a MIPS assembly language program

Computer operations

- More examples on translating C statements into MIPS assembly language
  - Consider the C statements

    a = b + c;

    d = a – e;
  - Translation in MIPS assembly language

    add a, b, c

    sub d, a, e
  - Notice the use of the instruction "sub"
  - Notice that we have implicitly assumed that the MIPS instruction operands have the same name as the C statement variables
    - This is usually not the case; we will clarify soon

Computer operations

- More examples on translating C statements into MIPS assembly language
  - Consider the same C statements written as

    d = (b + c) – e;
  - MIPS translation

    add d, b, c

    sub d, d, e
  - Consider the C statement

    f = (g + h) – (i + j);
  - MIPS translation may require storing the results of two additions in two temporary variables

    add t0, g, h

    add t1, i, j

    sub f, t0, t1

Computer operations

- Summary
  - Computer operations come in many types
    - We have discussed a few and will discuss many more
  - Each distinct operation has a corresponding instruction that the computer understands
  - Each operation has a few operands
    - Usually three in most computers
    - Two are source operands and one is destination operand
    - Accordingly, each instruction has a corresponding number of operands
    - In a computer, all instructions may have the same number of operands or may have a variable number of operands
      - Simple designs favor uniformity and regularity in instructions

Operands

- So far we have assumed that instruction operands are same as HLL program variables
  - This assumption is incorrect
  - Each hardware operation reads its source operands from some storage and writes its destination operand to some storage
    - This storage can be either a register from the register file inside the processor or a memory location
      - By memory we usually mean RAM of the computer (DRAM)
      - Recall that anything that is accessed from DRAM is also copied into SRAM cache inside the processor replacing something if the SRAM cache is already full
      - So, at any point in time, the SRAM cache stores a subset of the DRAM contents
      - The data in a memory location can be found in SRAM cache or in DRAM (DRAM is accessed only if SRAM cache does not have the data)

Operands

- Translating an HLL program to an assembly language program involves two basic steps
  - Mapping HLL operators to computer instructions
    - Exact instruction names differ from one computer to another, but the operations done by the instructions are largely same
  - Mapping HLL operands (or variables) to computer instruction operands
    - Requires deciding which variable is stored where and when
    - Every variable gets a fixed memory location
      - However, this forces every computer instruction to operate on memory operands only and obviates the need for a register file
      - But accessing a register is faster than accessing memory (faster than even SRAM cache)

Obviate: R

Operands

- Mapping variables to operands
  - Fast access to registers motivates mapping variables to register operands
  - Simplicity of design also motivates restricting the operands of certain instructions to registers only
    - Consistent latency of these operations (recall that accessing memory can have variable latency)
    - In MIPS, many instructions (including arithmetic) allow only register operands
    - In x86, both register and memory operands are allowed
      - Makes instruction latency variable
  - Width of register file dictates the width of a register operand (usually 32 or 64 bits)
    - Dictates the width of datapath used in computation

Operands

- Mapping variables to operands
  - Register width is typically decided based on the primitive variable types used in HLL programs
    - Makes it easy to map variables to registers
    - In C language, these are char, short, int, long long, float, double
    - Notice that 64-bit registers are large enough to hold a variable of any of these types
    - Since wider register file is slower, it is important to find out how often a 64-bit operand is used
      - If that is not too common, a 32-bit wide register file would suffice
      - A 64-bit operand will have to be mapped on a pair of registers and may require double the 32-bit access time
      - Important design principle: make the common case fast

Operands

- Mapping variables to operands
  - Since taller register file is slower, the number of registers must be restricted
  - Since each variable is given a unique memory location to store its value, it must be copied to a register if an instruction wants to use it as a register operand
    - In MIPS, arithmetic and many other instructions only allow register operands
  - Not all variables of a program can be allocated in registers at the same time due to restricted number of registers
    - Variables are allocated in and de-allocated from registers as the program progresses (filled from and spilled into memory)

Operands

- Mapping variables to operands
  - Register allocation of variables is the compiler's responsibility
    - Goal is to minimize the number of fills and spills because memory access is inefficient
    - Example: assume four registers r1, r2, r3, r4

    Consider the C statements:

    ```
    a=b + c; // Allocate a, b, c to r1, r2, r3
    d=e + f; // Allocate d to r4, how to allocate e and f?
    a=a + d;
    b=a + e;
    ```

Operands

- Mapping variables to operands

  Consider the C statements:

  ```
  a=b + c; // Allocate a, b, c to r1, r2, r3
  d=e + f; // Allocate d to r4, how to allocate e and f?
  a=a + d;
  b=a + e;
  ```

  - Assembly language translation (not exactly MIPS)

    ```
    load r2, addr_b     #fill b
    load r3, addr_c     #fill c
    add r1, r2, r3      #a = b + c
    load r2, addr_e     #fill e
    load r3, addr_f     #fill f
    add r4, r2, r3      #d = e + f
    add r1, r1, r4      #a= a + d
    add r3, r1, r2      #b=a + e
    store r1, addr_a    #spill a (note changed syntax)
    store r3, addr_b    #spill b
    store r4, addr_d    #spill d
    ```

Operands

- Mapping variables to operands
  - Large number of registers not only increase the access time of register operands, but also can increase the clock cycle time
    - Depends on how much work needs to be done within a clock cycle
    - Cycle time could be register read time + latency of combinational operation + setup time of register + register write time + clock skew time
      - Notice that in this formulation, the propagation delay of registers is not important
    - Register read, computation of the operation, and register write can be split into three consecutive cycles
      - The cycle time will decrease compared to the previous one, but will remain limited by max(read latency, op latency, write latency)

Operands

- Mapping variables to operands
  - MIPS assembly language denotes registers using the $ sign followed by the register name or number (e.g., $0, $1, …, $31 for 32 registers)
    - The book uses a different notation: allocates the program variables in registers $sN where N is an integer and any temporary variable in registers $tN
      - Makes sure that the total number of registers never exceeds 32 because MIPS instruction set defines only 32 registers
    - Consider compiling f = (g + h) – (i + j);
    - Assume that f, g, h, i, j are allocated to $s0, $s1, $s2, $s3, $s4

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

Operands

- Mapping variables to operands
  - For instructions that have two register sources and one register destination, the best performance is achieved if the register file has two read ports and one write port
    - Two reads can be done in parallel
    - If the register file had a single read port, this would have required two sequential accesses doubling the overall time to read the source operands
    - If we assume that one instruction executes at a time, the write port can share the decoder, wordline and the bitline with one of the read ports
      - This is called an RW port (supports both read and write)
      - This assumption is usually not true and we will lift this assumption later

Operands

- Fill and spill instructions in MIPS require memory and register operands
- Other ISAs (such as x86) allow even arithmetic instructions to have memory operands
- Memory operands essentially need to specify a memory address
  - Just like register operands specify a register name (which translates to a register number or register address)
  - Since the number of registers is small, it is easy to specify a register address (requires few bits)
    - Number of bits required: log of number of registers

Operands

- Memory operands essentially need to specify a memory address
  - Since instructions need to be stored in memory, it is important to be economical in specifying operands
  - Having 4 GB memory would require 32 bits of address to specify a memory operand
    - Not particularly attractive (called direct memory addressing mode)
    - Recall that instructions are generated by the compiler and the compiler does not always know the addresses of all variables of a program
      - Think about the return pointer of a malloc call (resolved only when the program runs)

Operands

- Memory operands essentially need to specify a memory address
  - Memory operand addresses are usually specified relative to a base register
    - 100($2) is interpreted as (100 + contents of $2) and the result is taken as the address of a memory operand
    - The instruction could be "load $1, 100($2)"
      - Fills register $1 with the contents at address (100 + $2)
    - Known as displacement mode of memory addressing
      - The constant 100 is called the displacement and can be positive or negative
      - The register $2 is known as the base register (can use any register as the base register)

Operands

- Displacement mode of memory addressing
  - Very useful in walking through an array or a structure
    - The starting address of the array can be stored in the base register and the displacement can be gradually incremented
    - The starting address of a structure can be stored in the base register and the displacement can specify the offset to the field of the structure to be accessed
  - This is the only supported mode of addressing memory in MIPS

Operands

- Memory operands are filled (or loaded) into registers before they can be operated on and the register contents can be spilled (or stored) to memory
  - Memory operands are represented by a memory address which refers to a byte and a register is typically bigger than a byte
    - Four- or eight-byte registers in most ISAs
    - Need to disambiguate this size mismatch
  - To resolve this, data transfer instructions also specify the number of bytes to be transferred between register and memory
    - Loads and stores come in different flavors in MIPS

Operands

- Memory operand size
  - MIPS has four different load and four different store instructions corresponding to four different size
    - Load byte: lb
    - Load half word: lh (loads two bytes)
    - Load word: lw (loads four bytes)
    - Load double word: ld (supported in 64-bit MIPS)
    - Store byte: sb
    - Store half word: sh
    - Store word: sw
    - Store double word: sd (supported in 64-bit MIPS)

Operands

- Memory operand size
  - In x86 ISA, an instruction may have multiple memory operands
    - Not possible to encode memory operand size in instruction name because different memory operands may have different sizes within the same instruction
      - Load a byte from address A, load a word from address B, add them, and store the result in a register
    - Each memory operand needs to specify its size separately within the instruction
      - Requires more bits in the instruction (e.g., could be two bits per memory operand)

Operands

- Sub-word memory operands in 32-bit MIPS
  - Consider the load byte or load half word instruction
    - These instructions load a byte or two bytes into the least significant side of the destination register
    - What about the remaining bits of the destination register?
      - Two options: zero-filled or sign-extended
      - Sign-extension fills the remaining bits with the sign bit of the loaded byte or the loaded half word
      - Notice that in two's complement representation, extending a number's sign bit by arbitrary number of bits does not change the equivalent decimal value of the number
      - Introduces two more flavors for sub-word loads
        - » Load byte unsigned (lbu) and load byte (lb); load half unsigned (lhu) and load half (lh);
        - » Unsigned flavor fills the remaining bits of the destination register with zeros; the other one sign-extends

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    long long x[10];

    long long y = x[2];
  - In 64-bit MIPS, the assignment statement will generate a ld instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate ld $1, 16($29)
    - Each element of x is eight bytes
  - In 32-bit MIPS, the assignment statement will generate two lw instructions
    - Assume that y is split into two registers $1 and $2

    lw $1, 16($29)

    lw $2, 20($29)

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    ```
    int x[10];
    int y = x[2];
    ```

  - The assignment statement will generate a lw instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate lw $1, 8($29)
    - Each element of x is four bytes

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    short x[10];

    short y = x[2];
  - The assignment statement will generate a lh instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate lh $1, 4($29)
    - Each element of x is two bytes

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    ```
    unsigned short x[10];
    unsigned short y = x[2];
    ```

  - The assignment statement will generate a lhu instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate lhu $1, 4($29)
    - Each element of x is two bytes

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    ```
    char x[10];
    char y = x[2];
    ```

  - The assignment statement will generate a lb instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate lb $1, 2($29)
    - Each element of x is one byte

Operands

- When does the compiler generate which load instruction?
  - Consider the following C code snippet

    ```
    unsigned char x[10];
    unsigned char y = x[2];
    ```

  - The assignment statement will generate a lbu instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate lbu $1, 2($29)
    - Each element of x is one byte

Operands

- Sub-word memory operands in 32-bit MIPS
  - Store byte and store half word do not have the unsigned or signed flavors
    - Store byte instruction takes the least significant byte from the source register (left operand) and stores the byte at the address of the memory operand

    sb $2, 0($1)

    - Store half word instruction takes the least significant two bytes from the source register (left operand) and stores the bytes starting at the address of the memory operand

    sh $2, 0($1)

    - There is no scope of zero-fill or sign-extension

Operands

- When does the compiler generate which store instruction?
  - Consider the following C code snippet

```
unsigned char x[10];
unsigned char y;
x[2] = y;
```

  - The assignment statement will generate a sb instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate sb $1, 2($29)
    - Each element of x is one byte
  - Changing type to "char" does not change the instruction
    - Bit#7 of y already encodes the sign (LSB is bit#0)

Operands

- When does the compiler generate which store instruction?
  - Consider the following C code snippet

```
unsigned short x[10];
unsigned short y;
x[2] = y;
```

  - The assignment statement will generate a sh instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate sh $1, 4($29)
    - Each element of x is two bytes
  - Changing type to "short" does not change the instruction
    - Bit#15 of y already encodes the sign (LSB is bit#0)

Operands

- When does the compiler generate which store instruction?
  - Consider the following C code snippet

```
unsigned int x[10];
unsigned int y;
x[2] = y;
```

  - The assignment statement will generate a sw instruction
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate sw $1, 8($29)
    - Each element of x is four bytes
  - Changing type to "int" does not change the instruction
    - Bit#31 of y already encodes the sign (LSB is bit#0)

Operands

- When does the compiler generate which store instruction?
  - Consider the following C code snippet

```
unsigned long long x[10];
unsigned long long y;
x[2] = y;
```

  - The assignment statement will generate a sd instruction in 64-bit MIPS
    - If y is allocated to $1 and the starting address of x is in $29, the compiler will generate sd $1, 16($29)
    - Each element of x is eight bytes
  - Changing type to "long long" does not change the instruction
    - Bit#63 of y already encodes the sign (LSB is bit#0)

Operands

- When does the compiler generate which store instruction?
  - Consider the following C code snippet

```
unsigned long long x[10];
unsigned long long y;
x[2] = y;
```

  - The assignment statement will generate two sw instructions in 32-bit MIPS
    - If y is split into $1 and $2 and the starting address of x is in $29, the compiler will generate the following

```
sw $1, 16($29)
sw $2, 20($29)
```

Operands

- Alignment of memory operands
  - MIPS requires that a memory operand of size n bytes must start at an address that is divisible by n
    - The least significant log(n) bits of the memory operand address must be zero
    - Observe that byte operands (n==1) can start at any address
      - This is a property of byte-addressable memory
      - Each memory address corresponds to a distinct byte
    - Observe that n can take only a few values: 1, 2, 4, 8
  - x86 ISA allows unaligned memory operands

Operands

- Alignment of memory operands
  - What can be the reason for this alignment requirement?
    - Observe that DRAM row sizes and interface between the SRAM cache and the DRAM controller are powers of two and are usually bigger than the largest n
      - Therefore, DRAM row sizes and SRAM cache to DRAM controller interface sizes are divisible by n
    - If a half word memory operand starts at an odd address, there is a chance that it may get split across two DRAM rows or two transactions between SRAM cache and DRAM controller
      - This will double the operand fill time because this will require activating two DRAM rows one after another or making two transactions to the DRAM controller
    - Similar arguments apply for word and double word operands

Operands

- Byte ordering within a memory word
  - Suppose we are loading a half word from address zero
    - We need to load two bytes from addresses 0 and 1
    - Which of these is the least significant byte?
      - Decided by how the bytes are ordered within a word i.e., how you assign addresses to bytes within a word
      - The word containing the half word to be loaded starts at address zero and ends at address 3; need to assign these four addresses (0, 1, 2, 3) to the bytes in a word
      - It is possible to do this assignment in 4! ways
      - Out of these 24 different orders, only two are sequential in nature; in one case, the most significant byte has address 0 and the address increases linearly till the least significant byte gets address 3 (called big endian ordering); in the other case, the least significant byte gets address 0 and the address increases linearly till the most significant byte gets address 3 (called little endian ordering)

Operands

- Byte ordering within a memory word
  - Endianness depends on where the address 0 is within a word
    - If address 0 is at the most significant byte (the big end), it is called big endian
    - If address 0 is at the least significant byte (the little end), it is called little endian
    - MIPS follows big endian ordering
    - x86 follows little endian ordering
  - It is important to know the endianness of a machine if you are trying to write program to communicate between two machines
    - If the machines have different endianness, extra effort needed on the receiving side

Operands

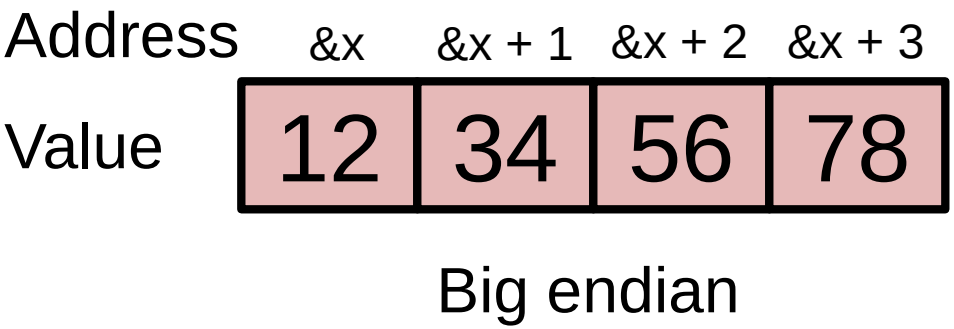
- Word ordering within a double word
  - Two words in a double word
    - Let the addresses of the words be zero and one
  - Big endian machines have word#0 as the most significant word and word#1 as the least significant word
  - Little endian machines have word#0 as the least significant word and word#1 as the most significant word

Operands

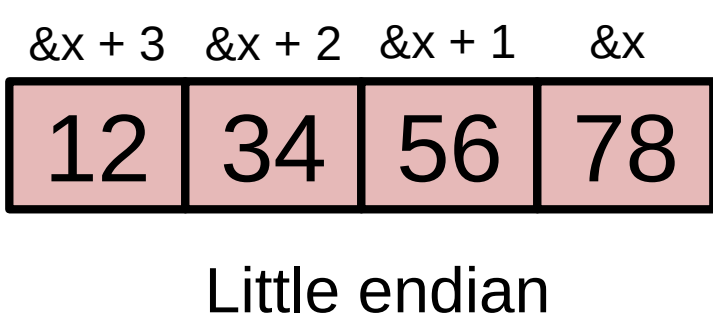
- Example C code to distinguish between big endian and little endian ISAs

```
unsigned int x = 0x12345678;
unsigned char y = *(char*)&x;  // byte at start address
printf("%#x\n", y);
```

  - A big endian machine would print 0x12
  - A little endian machine would print 0x78



Big endian



Little endian

Operands

- How to do unaligned memory operand accesses in MIPS?
  - Is it really important?
  - Suppose I would like to extract four consecutive characters starting from any arbitrary location of a string, say, "abcdefghij"
    - For example, I may want to extract "defg", which starts at an unaligned word address
      - One option is to do four load byte operations and then somehow put them together properly in a 32-bit register
        - » A lot of instructions needed
      - Slightly better option is to do two load bytes (for 'd' and 'g') and one load half word (for "ef")
        - » Still a lot of instructions needed
  - Supporting unaligned access is important for reducing the number of instructions in a program

Operands

- MIPS supports two unaligned load word instructions and two unaligned store word instructions
  - lwl (load word left)
  - lwr (load word right)
  - swl (store word left)
  - swr (store word right)

Operands

- Understanding lwl and lwr
  - Example: lwl $4, 0($10); let $10 contain 0x57
    - Let the word containing this byte address be w
    - Extract the bytes contained by w that *start* from this address (remember this is big endian)
    - Put these bytes (in this case one byte) in the *upper* portion of $4 and leave the remaining bytes (in this case 3 lower bytes) unchanged
  - Example: lwr $4, 0($10); let $10 contain 0x5a
    - Extract the bytes contained by w that *end* at this address
    - Put these bytes (in this case three bytes) in the *lower* portion of $4 and leave the remaining bytes (in this case the upper byte) unchanged

Operands

- When should the compiler generate lwl and lwr?
  - Suppose the char pointer x points to “abcdefghij” i.e., char x[10] = “abcdefghij”
  - If we want to extract “defg”, the compiler will generate an lwl from address x+3 and an lwr from address x+6

```
int y = *((int*)&x[3]);
```

Operands

- Handling constant operands
  - Compiling a = b + 3 where a and b are integers
    - Suppose a is allocated to $11, b is allocated to $12, and the constant 3 is allocated to $13
    - Suppose the address of a is 0($1), address of b is 0($2), and address of 3 is 0($3)

```
lw $12, 0($2)
lw $13, 0($3)
add $11, $12, $13
sw $11, 0($1)
```

  - Observe that allowing constant operands in instructions can eliminate loading constants from memory
    - These are called immediate operands

Operands

- Handling constant operands
  - The addi instruction allows one of the two source operands to be an immediate
    - The add immediate instruction
    - Compilation of a = b + 3 is the following

```
lw $12, 0($2)
addi $11, $12, 3
sw $11, 0($1)
```

  - Immediate operands can be negative also
    - There is no need to have a subi instruction
  - The constant zero is a very common operand
    - Also, copying a value from one register to another can be compiled as an addi instruction with zero as an immediate operand
    - MIPS dedicates register $0 to always hold the value 0

Operands

- Example uses of $0
  - How to compile a = 3 – b
    - Suppose a is allocated to $11, b is allocated to $12
    - Suppose the address of a is 0($1), address of b is 0($2)

```
lw $12, 0($2)
sub $12, $0, $12
addi $11, $12, 3
sw $11, 0($1)
```

    - Alternate compilation (requires one extra register)

```
lw $12, 0($2)
addi $13, $0, 3
sub $11, $13, $12
sw $11, 0($1)
```

Operands

- Summary
  - Three types of operands: register operands, memory operands, immediate operands
  - Register operands are represented by register names and register names have one-to-one mapping to register numbers
    - MIPS ISA has 32 distinct registers
    - Out of these, one register is hardwired to store zero
  - Memory operands use displacement addressing mode in MIPS
    - A memory operand address is represented as the sum of a base register content and a displacement
    - Displacement can be positive or negative
  - Immediate operands represent constants

Logical operations

- So far we have seen only arithmetic operations
  - Addition, Subtraction (add, sub, addi in MIPS)
  - Computers also support multiplication and division operations (mult, div in MIPS)
- Logical operations operate on bits of the operands
  - Operands can be either registers or immediates
  - Some of the MIPS logical instructions: and, or, nor, xor, andi, ori, xori, sll (shift left logical), srl (shift right logical), sra (shift right arithmetic), sllv (shift left logical variable), srlv, srav

Logical operations

- Bit-wise logical operators of C nicely map to the logical operations supported by MIPS
  - &, |, ^ map to and, or, xor
    - If one operand is immediate, andi, ori, xori are generated
  - MIPS does not have a "not" instruction
    - To maintain the uniformity in number of operands
  - The nor instruction can be used to compile ~
    - If one operand is zero, nor is equivalent to not

Logical operations

- Bit-wise logical operators of C nicely map to the logical operations supported by MIPS
  - >> maps to srl or sra or srlv or srav depending on the type of the operand being shifted and the shift amount
    - If the operand being shifted is of unsigned type, the srl or srlv instruction is generated
      - Zeros are shifted into the most significant side
      - If the shift amount is a constant and at most 31, the srl instruction is generated
      - If the shift amount is larger or a variable, the srlv instruction is generated where the shift amount is specified in a register
      - In both cases, shifting right by one bit position is equivalent to dividing by two (integer division)

Logical operations

- Bit-wise logical operators of C nicely map to the logical operations supported by MIPS
  - >> maps to srl or sra or srlv or srav depending on the type of the operand being shifted and the shift amount
    - If the operand being shifted is of signed type, the sra or srav instruction is generated
      - Sign bit is shifted into the most significant side so that the shifted result retains appropriate sign of the operand
      - If the shift amount is a constant and at most 31, the sra instruction is generated
      - If the shift amount is larger or a variable, the srav instruction is generated where the shift amount is specified in a register

Logical operations

- Bit-wise logical operators of C nicely map to the logical operations supported by MIPS
  - << maps to sll or sllv depending on the shift amount
    - Always zeros are shifted into the least significant side
    - Shifting by one bit position is equivalent to multiplying by two

Logical operations

- Example compilation
  - Suppose a, b, c are allocated to registers $11, $12, $13 and their memory addresses are in $1, $2, $3

```
int a, b, c;

a = b ^ c;
b = a & c;
a = c | a;
b = ~a;
b = b ^ 0x20;
a = b & 0x7;
```

```
lw $12, 0($2)
lw $13, 0($3)
xor $11, $12, $13
and $12, $11, $13
or  $11, $13, $11
nor  $12, $11, $0
xori  $12, $12, 0x20
andi $11, $12, 0x7
sw    $11, 0($1)
sw    $12, 0($2)
sw    $13, 0($3)
```

Logical operations

- Example compilation
  - Suppose a, b, c are allocated to registers $11, $12, $13 and their memory addresses are in $1, $2, $3

```
int a, b;
unsigned int c;
a = b >> 12;
c = c >> 4;
b = b >> c;
c = c >> b;
c = c*8;
b = a << c;
```

```
lw $12, 0($2)
lw $13, 0($3)
sra $11, $12, 12
srl $13, $13, 4
srav $12, $12, $13
srlv $13, $13, $12
sll  $13, $13, 3
sllv $12, $11, $13
sw    $11, 0($1)
sw    $12, 0($2)
sw    $13, 0($3)
```

Operations for making decision

- Interesting programs decide the execution path based on the computation results
  - In C language, these are expressed using if-else if-else blocks
  - Involves evaluating a condition and taking one of the two possible paths of execution
    - Fall through or jump
    - The basic operation used in evaluating a condition is comparison

```
if (x == y) z++;  // Fall through path
else z--;         // Jump path
```

    - MIPS ISA offers a set of instructions that can compare and jump based on the outcome of the comparison
      - Known as conditional branch instructions

Conditional branch instructions

- MIPS ISA comes with six flavors of such instructions
  - beq (Branch if equal) compares two register operands
  - bne (Branch if not equal) compares two register operands
  - bgez (Branch if >= 0) compares a register operand against zero
  - bgtz (Branch if > 0)
  - blez (Branch if <= 0)
  - bltz (Branch if < 0)
  - Enough to compile any condition and jump

Conditional branch instructions

- Sometimes it is necessary to make unconditional jumps
  - C programming language has “go to” for this purpose
  - MIPS ISA offers an unconditional jump instruction denoted by j

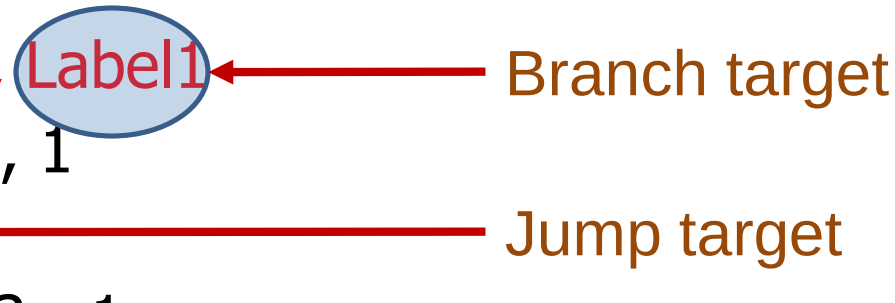
Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x == y) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        bne $11, $12, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```



Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x != y) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        beq $11, $12, Label1
        addi $13, $13, 1
         j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x <= 0) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        bgtz $11, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x < 0) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        bgez $11, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x >= 0) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        bltz $11, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x > 0) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        blez $11, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x > y) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        sub $14, $11, $12
        blez $14, Label1
        addi $13, $13, 1
         j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x > y) z++;
else z--;
```

    - Instead of a subtraction, a comparison operation can be invoked to save energy and possibly time
    - The instructions slt and slti achieve this
      - slt compares two source register operands and sets the destination register contents to one if the first operand is less than the second
      - slti compares a source register against an immediate operand and sets the destination register contents to one if the source register operand is less than the immediate

Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x > y) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        slt $14, $12, $11
        beq $14, $0, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if (x >= y) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        slt $14, $11, $12
        bne $14, $0, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x < 3) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        slti $14, $11, 3
        beq $14, $0, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x <= 3) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        slti $14, $11, 4
        beq $14, $0, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Example compilation
  - Suppose x, z are allocated to registers $11, $13 and their addresses are in $1, $3

```
int x, z;
if (x > 3) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $13, 0($3)
        slti $14, $11, 4
        bne $14, $0, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Signed and unsigned comparisons
  - Comparison between two 32-bit signed values and between two 32-bit unsigned values must be done differently
    - The interpretation of the most significant bit is different
    - For signed types, the entire 32-bit value must be treated as a two's complement value
      - The most significant bit will be multiplied by -2^{31}
    - For unsigned types, the entire 32-bit value must be treated as an unsigned binary value
      - The most significant bit will be multiplied by 2^{31}

```
unsigned int a = 0xFFFFFFF0;
int b = (int)a;  // What is the decimal value of b?
```

Conditional branch instructions

- Signed and unsigned comparisons
  - Consider the following C code

```
unsigned int a = 0xFFFFFFF0;
unsigned int b = 0xF;
int x = 0xFFFFFFF0;
int y = 0xF;
int c = 0;
if (a < b) c++;
else c--;
if (x < y) c++;
else c--;
```

  - Final value of c is zero; the two comparisons must be done differently
    - In one case, the comparison operands must be treated unsigned and in the other case, they are treated signed

Conditional branch instructions

- Signed and unsigned comparisons
  - Comparison between two 32-bit signed values and between two 32-bit unsigned values must be done differently
  - MIPS ISA offers two flavors of slt and slti to convey to the hardware how the operands of the comparison must be treated
    - sltu and sltiu instructions treat the comparison operands as unsigned values
    - slt and slti instructions treat the comparison operands as signed values
    - In the example of the last slide, the first comparison would generate the sltu instruction and the second would generate the slt instruction

Conditional branch instructions

- Compiling array bound check
  - Consider the following C code

```
int i, N;
if ((i < N) && (i >= 0)) a[i]++;
else go to EXIT
```

  - Suppose i is in $1, N is in $2, a is an integer array and its starting address is in $3, and a[i] is allocated in $13

```
sltu $4, $1, $2
beq $4, $0, EXIT
sll $4, $1, 2
add $4, $4, $3
lw $13, 0($4)
addi $13, $13, 1
```

Conditional branch instructions

- Compiling complex conditions
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if ((x == y) && (x >= 0)) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        bne $11, $12, Label1
        bltz $11, Label1
        addi $13, $13, 1
        j Label2
Label1: addi $13, $13, -1
Label2: sw $13, 0($3)
```

Conditional branch instructions

- Compiling complex conditions
  - Suppose x, y, z are allocated to registers $11, $12, $13 and their addresses are in $1, $2, $3

```
int x, y, z;
if ((x == y) || (x >= 0)) z++;
else z--;
```

- MIPS assembly language program:

```
        lw $11, 0($1)
        lw $12, 0($2)
        lw $13, 0($3)
        beq $11, $12, Label1
        bgez $11, Label1
        addi $13, $13, -1
        j Label2
Label1: addi $13, $13, 1
Label2: sw $13, 0($3)
```

Short circuit

Fall through and jump paths can be identified only after the C code is compiled

Conditional branch instructions

- Compiling complex conditions
  - Each clause is compiled from left to right
  - If conjunction of clauses is used, the compiled code looks for the first false clause so that no further evaluation of clauses is necessary
  - If disjunction of clauses is used, the compiled code looks for the first true clause so that no further evaluation of clauses is necessary
    - Converts the if block into the jump path
    - Fall through and jump paths cannot be correctly identified until the C code is compiled
  - This compilation procedure is known as short-circuiting and saves time on average

Conditional branch instructions

- Compiling loops
  - Consider the following while loop where a is an integer array (each element is four bytes)

```
while (a[i] == k) i++;
```

  - Assume that the starting address of array a is in $1, i is allocated to $2, k is allocated to $3, a[i] is allocated to $11, and address of i is in $12

```
START: sll $4, $2, 2
       add $4, $4, $1
       lw $11, 0($4)
       bne $11, $3, EXIT
       addi $2, $2, 1
       j START
EXIT:  sw $2, 0($12)
```

Conditional branch instructions

- Compiling loops
  - Consider the following while loop where a is an integer array (each element is four bytes)

    while (a[i] == k) i++;
  - Assume that the starting address of array a is in $1, i is allocated to $2, k is allocated to $3, a[i] is allocated to $11, and address of i is in $12
    - Optimized translation (why is it better than the previous?):

```
sll $4, $2, 2
add $4, $1, $4
START: lw $11, 0($4)
       bne $11, $3, EXIT
       addi $2, $2, 1
       addi $4, $4, 4
       j START
EXIT:  sw $2, 0($12)
```

Conditional branch instructions

- Compiling "for" loops
  - Consider the following "for" loop where a is an integer array (each element is four bytes)

    for (i=0; i<N; i++) a[i]++;
  - Suppose i is in $1, N is in $2, the starting address of array a is in $3, and a[i] is allocated in $13

```
        xor $1, $1, $1
        blez $2, EXIT
START:  lw $13, 0($3)
        addi $13, $13, 1
        sw $13, 0($3)
        addi $3, $3, 4
        addi $1, $1, 1
        slt $4, $1, $2
        bne $4, $0, START
EXIT:   …
```

Conditional branch instructions

- Conditional branches do not have any destination operand
  - At most two source operands for comparison
    - Two source register operands in beq and bne
    - One source register operand in blez, bltz, bgez, bgtz
  - Branch target is a constant and is represented as an immediate operand
    - Branch target or the label used in a branch instruction is the memory address where the target instruction is stored
    - Compiler usually does not know instruction addresses because these get fixed at a later stage
    - The branch target in a conditional branch instruction is encoded as how many instructions away the target is from the branch instruction (often called "offset")

Conditional branch instructions

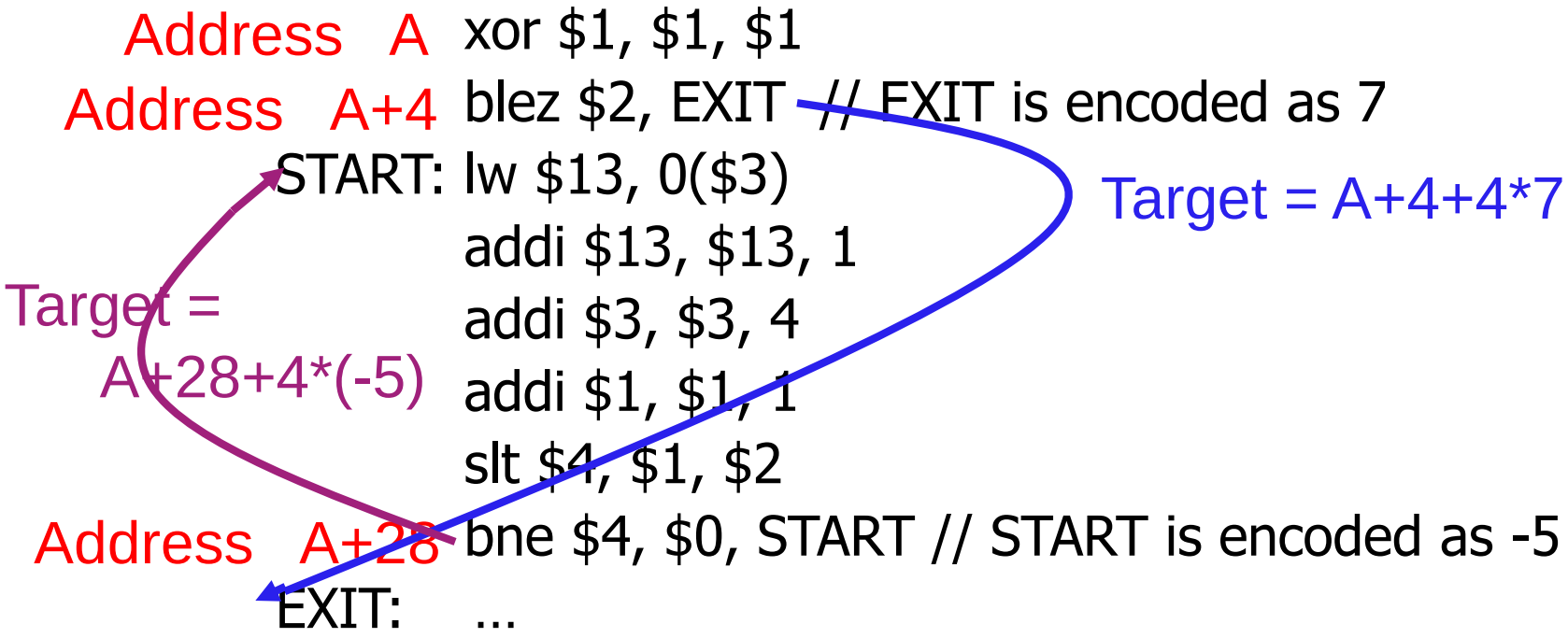
- Representing targets in conditional branches
  - Consider the following translation of a "for" loop

```
                xor $1, $1, $1
Forward branch  blez $2, EXIT  // EXIT is encoded as 7
        START:  lw $13, 0($3)
                addi $13, $13, 1
                addi $3, $3, 4
                addi $1, $1, 1
                slt $4, $1, $2
Backward branch bne $4, $0, START // START is encoded as -5
        EXIT:   …
```

  - A conditional branch with a positive offset is known as a forward branch; otherwise it is called a backward branch

Conditional branch instructions

- Representing targets in conditional branches



  - Target for a conditional branch instruction is computed by adding the offset multiplied by instruction size to the branch instruction's address
    - Works only if all instructions have the same size
    - All MIPS instructions have the same size (four bytes)

Compiling switch/case

- The successful case is not fixed

```
for (i=0; i<N; i++) {
   switch (x[i]) {
      case 0: …
      case 1: …
      …
      default: …
   }
}
```

- One possibility is to treat the chain of cases as if-else if-else chain
  - The number of comparisons depends on the position of the successful case in the chain
  - There can be a lot of wasted comparisons

Compiling switch/case

- A better option is to prepare a table where each entry is a case label and index into the table is the case value
  - In the last slide's example, let's refer to this table by the array T (usually known as jump table)
    - T[x[i]] should store the target case label
    - Compiler is responsible for populating this table with the correct case labels
  - Explains why only integer or character type is allowed in switch argument
  - The compiler must generate an instruction to jump to the correct label
    - This is an unconditional jump with a variable target
    - The j instruction is allowed to use a constant target

Compiling switch/case

- The code will be transformed by the compiler

```
for (i=0; i<N; i++) {
   switch (x[i]) {
      label0: case 0: …
      label1: case 1: …
      …
      labelD: default: …
   }
}
```

  - Transformed code:

```
for (k=0; k<JTSize; k++) T[k] = labelD
T[0] = label0; T[1] = label1; …
for (i=0; i<N; i++) goto T[x[i]];
```

- The compiler must generate an instruction to jump to the correct label (for goto T[x[i]])
  - This is an unconditional jump with a variable target
  - The j instruction is allowed to use a constant target

Compiling switch/case

- MIPS ISA provides an instruction to make unconditional jumps to variable targets
  - This instruction is jr (jump register)
    - Has one register operand holding the jump target
    - Compiler is responsible to generate code for loading the correct target in the register before the jr instruction

    jr $20

    - Takes the contents of $20 and treats it as a jump target
      - This target is NOT an offset relative to the address where the jr instruction is stored
      - This target is the absolute final target

Compiling switch/case

- Consider the following C code

```
switch (x[i]) {
  case 0: …
  case 1: …
  …
  default: …
}
```

  - Suppose jump table starting address is in $1, starting address of x is in $2, i is in $3, x is an integer array, and x[i] is allocated in $12

```
sll $4, $3, 2
add $4, $4, $2
lw $12, 0($4)
sll $12, $12, 2
add $4, $12, $1
lw $4, 0($4)    // Each jump table entry is 32 bits
jr $4
```

Compiling switch/case

- MIPS translation shown in the last slide is not particularly efficient
  - This translation works if we know the range of values x[i] can take
  - Wastes a lot of memory by preparing a jump table for the entire range of x[i]
  - Jump table size can be optimized by doing a simple range check on x[i] before using the jump table entry indexed by x[i]
    - All values of x[i] outside the range of case labels can be mapped to one jump table entry representing the default target

Using if-else

Compiling switch/case

- If case labels are not consecutive, a lot of jump table space is wasted
  - A simple range check on x[i] is not enough
- Compiling switch/case is fundamentally a search problem in general
  - Need to search for x[i] in the jump table and pick the corresponding case target
  - Any efficient search technique can be used
    - Hash tables are particularly attractive due to low average search time

Procedure/Function call

- Calling and returning from a function involves six steps
  - Caller puts the function parameters at appropriate places so that the function (sometimes referred to as callee) can access them
  - Jump to the first instruction of the function
  - Acquire portions of memory required for executing the function
  - Execute the instructions of the function
  - Put return values at appropriate places so that the caller can access them after return
  - Jump back to the origin of call

Procedure/Function call

- Where to put the function arguments?
  - MIPS function calling convention recognizes that the registers are the fastest storage options
  - It specifies four registers for passing function arguments
    - Usually denoted by $a0, $a1, $a2, $a3
    - Same as $4, $5, $6, $7
  - If a function has more than four arguments
    - First four are passed in registers
    - Rest of the arguments are spilled to memory and the function will have to fill them in registers before it can operate on them
      - Recall that the arithmetic and logic instructions in MIPS allow only register or immediate operands

Procedure/Function call

- How to jump to the function?
  - This is an unconditional jump with a constant jump target
  - Jumping to the function requires an additional operation
    - Remembering where to come back and start execution after the function completes
  - MIPS ISA offers an instruction called jump and link (jal) to carry out both operations
    - jal target
    - Target is usually the address of the first instruction of the function
    - Jumps to target and saves the address of the next instruction in $ra (stands for return address and same as $31)

Procedure/Function call

- Instruction addresses have a special name
  - Called program counter (PC)
  - Every instruction has a unique PC
  - If an instruction K has the PC equal to x, the next instruction would have PC x+s where s is the size of the instruction K in bytes
  - In MIPS, s=4 for all instructions
    - PC increases in steps of four
- Suppose the PC of a jal instruction is x
  - The PC of the instruction where the function should return to is x+4
  - The jal instruction will store x+4 in $31

Procedure/Function call

- How to jump to a function?
  - Assume that the PC of the first instruction of the function is y
  - The generated instruction for jumping to the function is "jal y"
    - The target is not exactly y, but for now we will assume that
    - Assume that the PC of the jal instruction is x
    - Stores x+4 in $31
  - The same function can be called from different places in the program
    - Each call will be translated to a corresponding jal instruction
    - Each jal instruction will store a different return address in $31

Procedure/Function call

- How to jump to a function?

a: f() → jal x ($ra is filled with a+4)

…

b: f() → jal x ($ra is filled with b+4)

…

…

c: f() → jal x ($ra is filled with c+4)

x: f() {

…

…

}

  - The jal instruction not only jumps to the function, but also establishes a link between the function and the caller by saving the return address

Procedure/Function call

- Where to put the return value of the function?
  - This needs to be fixed so that the caller can access and use the return value
  - Before the function returns, the return values are stored in $v0 and $v1 (same as $2 and $3)
  - Usually, $v0 would be used only since only one value can be returned from a function
  - In 32-bit MIPS, if a function returns a 64-bit value, the return value is split into two 32-bit values and placed in $v0 and $v1
    - The caller can figure out whether both registers contain the return value or only $v0 contains the return value by looking at the type of the return value
    - $v0 contains the most significant word for 64-bit value

Procedure/Function call

- How to jump back to the origin of the call?
  - Recall that the jal instruction before jumping to the function saved the return address in $31
  - Jumping back to the call site is accomplished with a single instruction: jr $31
    - This instruction ends the function
  - Inside the processor, a register stores the next instruction's program counter
    - The content of this register is used to fetch the next instruction from memory
  - The "jr $31" instruction copies the contents of $31 into the program counter register
  - In general, all branch and jump instructions copy the target into the program counter register

Procedure/Function call

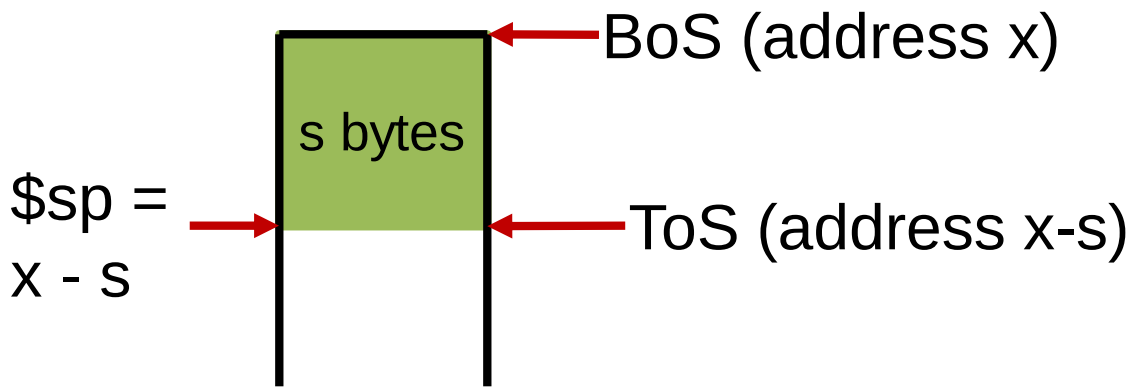
- How to acquire and release memory required by a function?
  - A function receives its first four parameters in four registers
  - The remaining parameters are spilled into memory
  - Additionally, the function may require some memory space for spilling registers
    - Happens when the compiler runs out of registers while compiling the function
    - In such situations, it must free up some registers, but it cannot just overwrite these registers
    - These registers are first saved in memory and later before returning, these registers are restored by copying them back from memory (caller may need the old values)

Procedure/Function call

- How to acquire and release memory required by a function?
  - The memory region usable by a function for spilling registers or function parameters belongs to a larger memory region known as the stack
  - After compiling a function, the compiler estimates how much stack area the function will require
    - This much stack area is reserved for the function
  - Spilling on to the stack is called pushing on the stack
    - Requires adjusting the top of stack pointer (or simply stack pointer) so that the next push doesn't overwrite the value at the top of stack
    - The stack pointer is maintained in $sp (same as $29)

Procedure/Function call

- How to acquire and release memory required by a function?
  - The stack grows downward meaning from higher address toward lower address
    - Pushing a 32-bit register requires decreasing $sp by four so that $sp now holds the address of the top of the stack
    - Initially, $sp holds the address of the bottom of the stack



Procedure/Function call

- How to acquire and release memory required by a function?
  - No special instruction for pushing on to the stack
    - Spilling to stack is accomplished through a store instruction to 0($sp)

```
sw $4, -4($sp)  // Spills $4 to stack
addi $sp, $sp, -4  // Adjusts $sp
sb $5, -1($sp)  // Spills only the least significant byte of $5
addi $sp, $sp, -1 // Adjusts $sp
```

    - Filling from stack is accomplished through load instructions

```
lb $5, 0($sp)   // Restores $5
lw $4, 1($sp)   // Restores $4
```

  - Freeing a portion of the stack is accomplished by increasing $sp by an appropriate amount
    - Known as popping the stack

Procedure/Function call

- How to acquire and release memory required by a function?
  - The stack region allocated to a function is popped at the end of the function

    addi $sp, $sp, positive_constant // compiler knows the value

    - This restores $sp to the point where it was before the function is called
    - This is important for the caller function to work correctly after the callee function returns
      - In C programming language, a function can be called only by a function, except for the main function which is called by a startup code that is attached with every program during linking

Procedure/Function call

- Example

```
int example (int g, int h, int i, int j) {
   return ((g+h) – (i+j));
}
int main (void) {
   int x = example (12, 1, 34, 42);
   return 0;
}
```

- MIPS translation

```
example: add $a0, $a0, $a1   // g+h
         add $v0, $a2, $a3   // i+j
         sub $v0, $a0, $v0
         jr $ra
main:    addi $sp, $sp, -4  // creates space for 32 bits on stack (used later)
         addi $a0, $0, 12
         addi $a1, $0, 1
         addi $a2, $0, 34
         addi $a3, $0, 42
         sw   $ra, 0($sp)  // Use the created stack space for saving $ra
         jal example        // This instruction overwrites $ra
         add $v0, $0, $0
         lw $ra, 0($sp)
         addi $sp, $sp, 4
         jr $ra
```

Procedure/Function call

- Example
  - In this example, the "example" function does not require any stack space and the "main" function requires just four bytes on stack to save $ra before calling "example"
  - More complex functions may require more space on the stack if they use a lot of variables
    - For example, if the "example" function has five parameters and if the fifth parameter is an integer, the main function would reserve eight bytes on stack for it
      - Four bytes would be used to save $ra before calling "example"
      - Four bytes would be used to store the fifth parameter of "example"

Procedure/Function call

- Example (assume that "example" has five parameters all of integer type)
  - MIPS translation

```
main:    addi $sp, $sp, -8  // Just one instruction to adjust $sp
         addi $a0, $0, 12
         addi $a1, $0, 1
         addi $a2, $0, 34
         addi $a3, $0, 42
         addi $v0, $0, 9    // Fifth parameter value is 9
         sw   $ra, 4($sp)
         sw   $v0, 0($sp)
         jal example
         add $v0, $0, $0
         lw $ra, 4($sp)
         addi $sp, $sp, 8
         jr $ra
example: lw $1, 0($sp)
         …
```

Procedure/Function call

- MIPS register saving convention
  - A function can use $8 to $25 for allocating variables
    - $8 to $15 and $24 and $25 are referred to as temporary registers ($t0 to $t7 and $t8 and $t9)
    - $16 to $23 are referred to as saved registers ($s0 to $s7) meaning that the function is supposed to save the contents of these registers on stack before using them and restore them after using them
      - Also referred to as callee saved registers
      - $sp and $ra are also callee saved registers
      - The contents of callee saved registers are preserved across calls because the called function saves and restores them
    - $t0 to $t9 are not preserved across calls
      - Caller must save them on stack before calling a function if the caller wants the contents of any of these to be preserved
      - Referred to as caller saved registers

Procedure/Function call

- MIPS register saving convention clearly specifies which registers are saved by the caller and which are saved by the callee
  - Helps minimize the number of spills and fills
  - If the caller wants the contents of any of the caller saved registers to be preserved, it must spill them to stack before calling a function
    - $t0 to $t9, $a0 to $a3, $v0, $v1
    - The callee does not worry about saving any of these registers; it can directly use them
  - If the callee wants to use any of the callee saved registers, it must save them to stack before using them and restore them from stack after using them
    - $s0 to $s7, $sp, $ra
    - The caller does not worry about saving any of these before calling a function

Procedure/Function call

- MIPS register saving convention clearly specifies which registers are saved by the caller and which are saved by the callee
  - A function first uses $t0 to $t9, $a0 to $a3, $v0, $v1 for allocating its variables before trying to use $s0 to $s7
    - Using $s0 to $s7 requires spilling them to stack and restoring them back
    - To minimize spills, it is always better to first exhaust the caller saved register set i.e., $t0 to $t9, $a0 to $a3, $v0, $v1

Function

Procedure/Function call

- Example: compiling recursive function

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

  - MIPS translation:

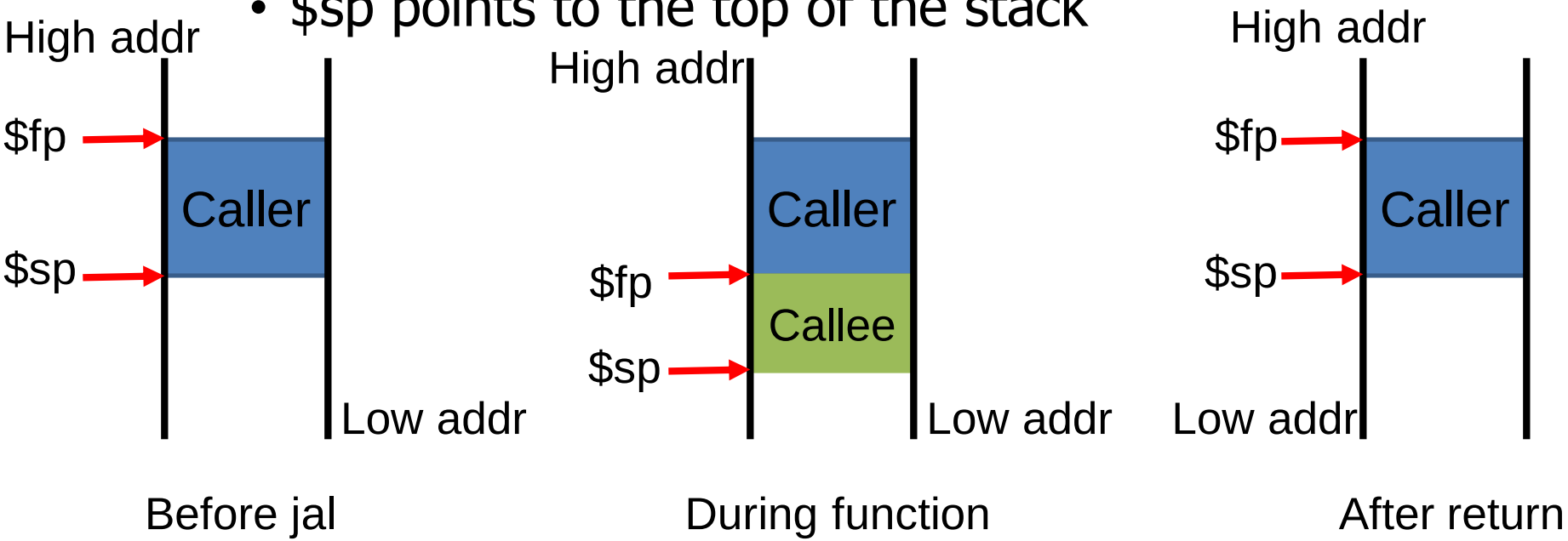
```
factorial: slti $t0, $a0, 1
           beq $t0, $0, Label
           addi $v0, $0, 1
           jr $ra
     Label: addi $sp, $sp, -8
           sw $ra, 4($sp)
           sw $a0, 0($sp)
           addi $a0, $a0, -1
           jal factorial
           lw $a0, 0($sp)
           mult $v0, $a0, $v0  // Not a correct MIPS instruction
           lw $ra, 4($sp)
           addi $sp, $sp, 8
           jr $ra
```

Procedure/Function call

- Functions in C access two types of variables
  - Automatic: local to function and discarded when the function returns
  - Static: global variables and anything declared as static; persist across calls and returns
  - All automatic variables are allocated on stack
    - Accessed using appropriate offsets relative to $sp
    - When a function returns, $sp is incremented so that all automatic variables of that function are popped
  - All static variables are allocated to the static or global region of memory
    - Accessed using appropriate offsets relative to $gp (same as $28)
    - $gp is a callee saved register

Procedure/Function call

- Stack portion created for a function's automatic variables is called a procedure frame or activation record
  - MIPS ISA defines a register known as frame pointer ($fp or $30) which can be used to record the bottom of the procedure frame
    - $sp points to the top of the stack

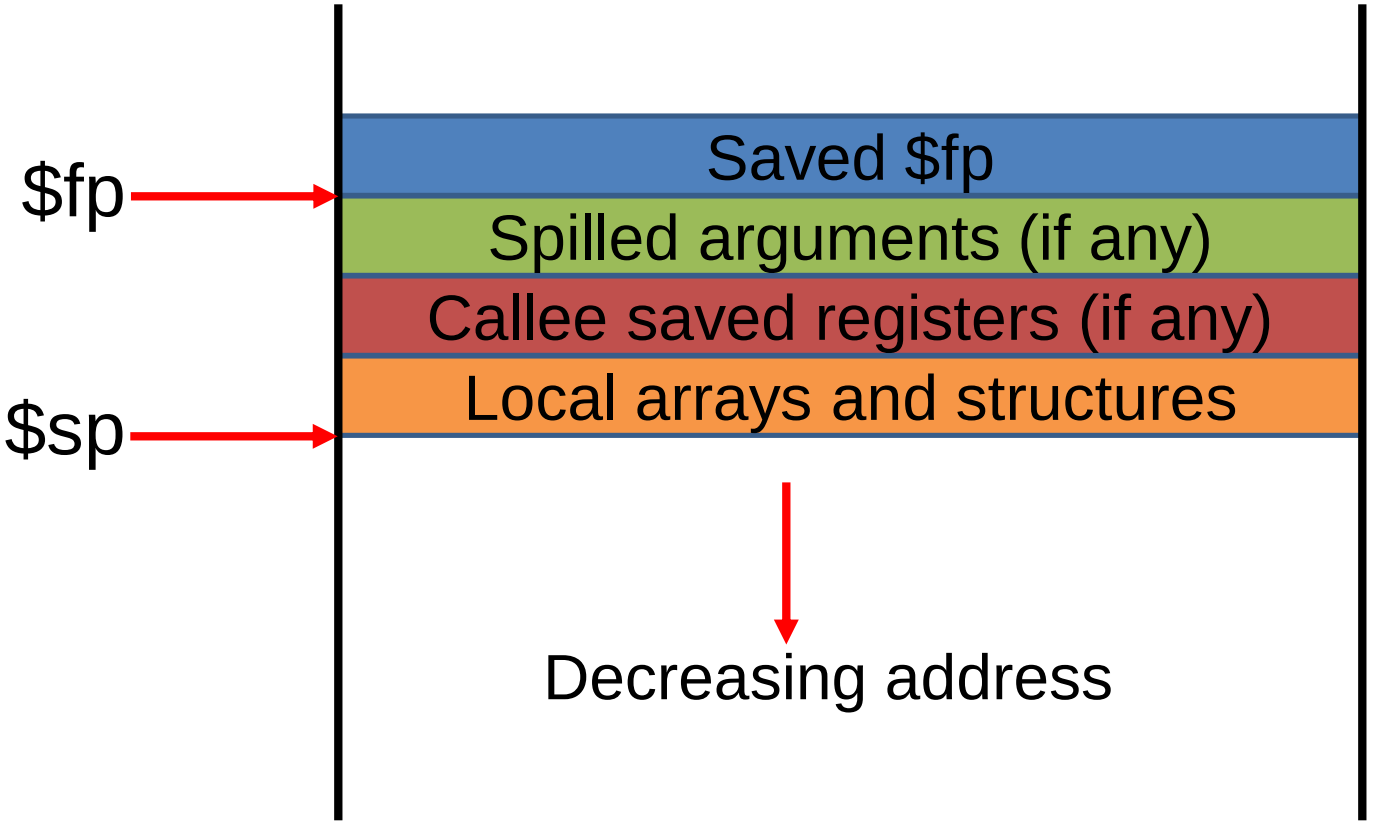


Procedure/Function call

- Why frame pointer is needed?
  - All automatic variables can already be accessed using displacements relative to $sp
  - $fp offers another alternative where all automatic variables can be accessed using displacements relative to $fp
    - Advantage is that $fp does not change during the function, but $sp can be adjusted from time to time during the function to push or pop variables
    - Assembly language functions written using $sp can be hard to understand due to the changing values in $sp
      - The same variable pushed on the stack at the beginning of a function can be referred to as d($sp) at one point in the function and as d'($sp) at another point in the function
      - This variable referred to relative to $fp will have the same displacement throughout the function because $fp always points to the bottom of the procedure frame

Procedure/Function call

- Structure of procedure stack frame



  - Before $sp is decremented by frame size, $fp is saved at 0($sp) and then $fp is set to $sp-4

Procedure/Function call

- Frame pointer is not used by some compilers
  - MIPS C compiler does not use $fp for pointing to the bottom of the procedure frame
    - Sacrifices readability of code
  - Instead, $fp is used as a regular callee saved register to expand the set of registers that can be used for allocating variables
    - Reduces number of spills to improve performance

Procedure/Function call

- So far we have assumed that a function call has a constant target known to the compiler
  - These are known as direct calls
- C programming language allows function pointers
  - A function pointer can point to any legitimate function
  - The value of a function pointer is the PC of the first instruction of the function it points to
  - During the execution of a program, a function pointer can be assigned different values
    - Just like any other pointer variable

Procedure/Function call

- Example use of function pointer

```
int f (int x, int y) {
   return x+y;
}
int g (int x, int y) {
    return x-y;
}
int main (void) {
   int (*fptr)(int, int);
   int a, b, c;
   scanf("%d %d", &a, &b);
   if (a > b) fptr = g;
   else fptr = f;
   c = fptr(a, b);
   printf("%d\n", c);
   return 0;
}
```

Target unknown at compile time

Procedure/Function call

- Calls made using function pointers are known as indirect calls
- Compiling indirect calls requires something like jr instruction with the linking facility so that the return address can be saved
- MIPS ISA offers the jump and link register (jalr) instruction for compiling indirect calls
  - Takes a register operand and uses the content of the register as the call target: jalr $20

Procedure/Function call

- Translating indirect calls
  - Consider the following code snippet

```
if (a > b) fptr = g;
else fptr = f;
c = fptr(a, b);
```

    - Suppose f starts at PC x and g starts at PC y (these constants are known at compile time)
    - Suppose fptr is in $11, a is in $12, and b is in $13
    - MIPS translation

```
        slt $t0, $13, $12
        beq $t0, $0, label
        la $11, y           # pseudo-instruction (load label address)
        j label1
label:  la $11, x           # pseudo-instruction (load label address)
label1: add $a0, $0, $12
        add $a1, $0, $13
        addi $sp, $sp, -4
        sw $ra, 0($sp)
        jalr $11
        …
```

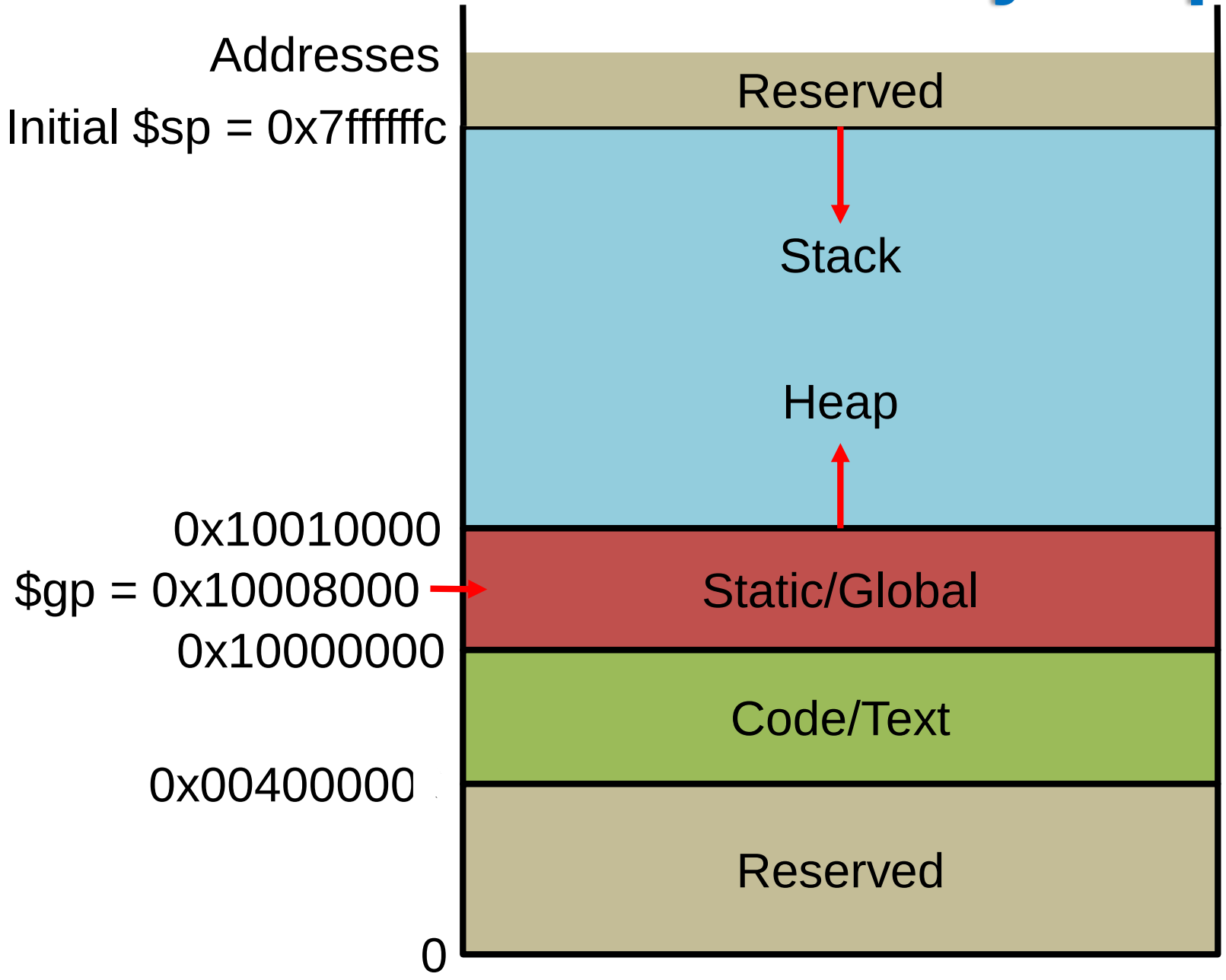
Procedure/Function call

- Data allocated dynamically within a function resides in a portion of memory known as the heap
  - Any pointer to heap data would still be allocated in registers if the pointer is local to the function
    - Local pointers are treated as automatic variables and can be spilled to stack if the compiler is short of registers
    - Local pointers cannot be used outside the function because the registers allocated to them may get overwritten by the caller after the function returns

32-bit MIPS memory map

- A program's memory has four regions: code (also called text), static/global data, heap data, stack data
  - Code region stores the instructions of the program and starts at address 0x00400000
  - Above code region is the static/global region starting at address 0x10000000
    - All data in this region are accessed relative to $gp which is initialized to the mid-point (address 0x10008000) of the static/global region
    - The displacement used with $gp can be positive or negative such that any data from address 0x10000000 to 0x1000ffff can be accessed

32-bit MIPS memory map



Instruction encoding

- A program is compiled into a sequence of instructions and is stored in a binary file
  - Requires each instruction to be encoded in binary (machine language)
- Before a program can start running at least part of this binary needs to be loaded in DRAM from the binary file
  - The file usually resides in a non-volatile storage medium such as the hard disk
  - The remaining parts can be loaded in DRAM as and when needed
- How to encode instructions in binary?

Instruction encoding

- Each part of an instruction must be encoded in binary
  - The instruction opcode and the operands
  - The operands can be addressing registers or memory location or can be constants
  - MIPS ISA allows only loads and stores to have memory operands
    - Requires encoding a signed displacement in addition to the base register
  - MIPS ISA allows at most one operand to be a constant or immediate
    - Is it useful to have more than one immediate operand?

Instruction encoding

- Each part of an instruction is referred to as a field
  - Fields to encode two source register addresses and a destination register address are required
    - Each of these is five bits wide because MIPS ISA has 32 registers
  - Width of the opcode field depends on the number of different types of instructions
    - MIPS ISA dedicates six bits to the opcode field: 64 different instructions
    - Additionally, there is a six-bit wide function field that is used to encode subtypes of instructions with opcode zero
      - Opcode zero is used for a subset of arithmetic and logic instructions that have no immediate operands

Instruction encoding

- Arithmetic and logic instructions execute in the arithmetic-logic unit (ALU)
  - A subset of instructions that have opcode zero (the function value is shown in parentheses)
    - sll (0x0), srl (0x2), sra (0x3), sllv (0x4), srlv (0x6), srav (0x7), jr (0x8), jalr (0x9), mfhi (0x10), mthi (0x11), mflo (0x12), mtlo (0x13), mult (0x18), multu (0x19), div (0x1a), divu(0x1b), add (0x20), addu (0x21), sub (0x22), subu (0x23), and (0x24), xor (0x26), nor (0x27), or (0x25), slt (0x2a), sltu (0x2b)
    - These are ALU instructions that do not have any immediate operands
    - Difference between add and addu instructions: addu instruction does not catch any overflow, but add does
      - Nothing to do with whether the operands are signed or unsigned (same is true for sub and subu)

Instruction encoding

- Arithmetic and logic instructions execute in the arithmetic-logic unit (ALU)
  - Multiplications are done through mult and multu instructions
    - Difference between mult and multu is that multu treats operands as unsigned; none detects overflow
    - These instructions do not have any explicit destination register and accepts only two source register operands
    - Two special registers called Hi and Lo store the most significant and the least significant words of the 64-bit result in 32-bit MIPS

```
mult $10, $20  // Multiplies $10 by $20 and sends the result to
               // Hi and Lo registers
```

Instruction encoding

- Arithmetic and logic instructions execute in the arithmetic-logic unit (ALU)
  - Integer divisions are done through div and divu instructions
    - Difference between div and divu is that divu treats the operands as unsigned; no overflow in division
    - These instructions do not have any explicit destination register and accepts only two source register operands
    - Two special registers called Hi and Lo store remainder and quotient respectively

```
div $10, $20  // Divides $10 by $20 and stores remainder in Hi
              // quotient in Lo
```

  - Floating-point multiplication and division have separate instructions

Instruction encoding

- Arithmetic and logic instructions execute in the arithmetic-logic unit (ALU)
  - The Hi and Lo registers cannot be used as operands of any ALU instructions
  - To further process the results of multiplication and division, the Hi and Lo register contents must be moved to general purpose registers
  - MIPS ISA offers two instructions to achieve this
    - mfhi moves the contents of Hi to a specified register
      - mfhi $t0
    - mflo moves the contents of Lo to a specified register
      - mflo $t6
  - There are matching instructions to move to Hi and Lo also: mthi $X and mtlo $Y (rarely used)

Instruction encoding

- C language disables overflow detection
  - MIPS C compiler always generates addu, subu
    - Results are same as add, sub except that nothing additional happens on overflow
- For multiplication involving signed operand types, mult is generated; otherwise multu is generated
- For division involving signed operand types, div is generated; otherwise divu is generated
- If overflow needs to be detected for mult and multu, it must be done in software
  - MIPS doesn't have any hardware support for detecting multiplication overflow

Instruction encoding

- Other than opcode and function fields, three fields are needed for specifying two source and one destination register operands
  - These three fields can be five bits each because MIPS has 32 registers
- To encode constant shift amounts (at most 31), we need a five-bit field

- All instructions conforming to this format are called R-format instructions (opcode is zero)
  - rs and rt are source register operands and rd is the destination register operand

Instruction encoding

- Not all fields are used by all instructions
  - Non-shift instructions and variable shift instructions do not use sh amt
  - sll, srl, and sra do not use rs
  - jr does not use rt, rd
  - jalr does not use rt (rd is always set to 31)
  - mfhi and mflo do not use rs, rt
  - mthi and mtlo do not use rt, rd
  - mult, multu, div, divu do not use rd
  - Unused fields are encoded as zero
- Length of these instructions could be shortened, but MIPS designers wanted all instructions to have a fixed four-byte size
  - Simple design, even though wastes memory

Instruction encoding

- R-format instructions have only register operands
- Instructions having one immediate operand are encoded using the I-format
  - Observation: these instructions require at most two register operands
  - Both register operands could be source operands: beq, bne, all store instructions
  - One register operand could be source and the other could be destination: addi, addiu, andi, ori, slti, sltiu, xori, all load instructions, lui
  - Some I-format instructions use only one register operand as a source: blez, bgtz, bltz, bgez

Instruction encoding

- The immediate operand is treated differently for different types of I-format instructions
  - Arithmetic instructions: signed immediate operand (addi, addiu, slti) or unsigned immediate operand (sltiu)
  - Logical instructions: unsigned immediate operand (andi, ori, xori, lui)
  - Branch instructions: PC-relative signed jump offset (beq, bne, blez, bltz, bgtz, bgez)
  - Load and store instructions: signed displacement from the base register

Instruction encoding

- I-format instructions having one destination register operand use rt as the destination
- I-format instructions having only one source register operand use rs as the source
- The immediate operand is sign-extended to 32 bits for addi, addiu, slti, sltiu (treats the sign-extended value as unsigned), all branches, all loads and stores
- The immediate operand is zero-extended to 32 bits for all logical instructions

Instruction encoding

- How to load a constant bigger than 16 bits into a register?
  - Suppose we would like to load the value 0x123456 into $10
  - Make use of the lui instruction (load upper immediate)

    lui $10, 0x12

    ori $10, $10, 0x3456

Instruction encoding

- Another use of lui: synthesizing addresses
  - Suppose the compiler wants to generate instructions to load the word starting at address 0x789abc into register $10

```
lui $1, 0x78          // $1 is also referred to as $at
ori $1, $1, 0x9abc // $at is a register used by assembler
lw $10, 0($1)
```

  - Note that the last two instructions cannot be replaced by lw $10, 0x9abc($1)
  - If the address was 0x345678, the following would work

```
lui $1, 0x34
lw $10, 0x5678($1)
```

Instruction encoding

- J-format is used to encode j and jal

- A 32-bit target is created as follows
  - The jump_target is shifted left by two bits to make it a four-byte aligned address
  - The most significant four bits of the PC of the jump instruction are concatenated in front of the shifted jump_target
- For conditional branches (I-format), the offset is shifted left by two bits, sign-extended, and added to PC to compute the branch target

Instruction encoding

- 16-bit branch offset in conditional branches also limits the distance to the branch target
  - If the branch target is too far away from the branch instruction, an unconditional jump can be used along with the conditional branch

```
if (x == y) { … }  // Problem: very large if block
else { … }
```

  - MIPS translation (incorrect)

```
        bne $t0, $t1, else_label
        …                          ≥ 2^15 instructions
        j label
else_label: …
label: …
```

Instruction encoding

- 16-bit branch offset in conditional branches also limits the distance to the branch target
  - If the branch target is too far away from the branch instruction, an unconditional jump can be used along with the conditional branch

```
if (x == y) { … }  // Problem: very large if block
else { … }
```

  - MIPS translation (correct)

```
                beq $t0, $t1, if_label
                j else_label
if_label:       …
                j label
else_label: …
label: …
```

Instruction encoding: Summary

- All instructions are 32-bit long
- Three formats distinguished by opcode
  - R-format: opcode is zero [op:rs:rt:rd:shamt:func]
    - Only register operands and shift amount
    - Also known as ALU format or special format
    - A very special R-format instruction: all zero (sll $0, $0, 0), known as NOP (no operation)
  - J-format: opcode is j (2) or jal (3) [op:target]
  - I-format: all other opcodes [op:rs:rt:immediate]
    - Instructions with immediate operand
    - Immediate is sign-extended in all instructions except logic instructions and lui
- Unused fields in an instruction are set to zero

Instruction encoding: Summary

- Three formats taken together define five operand addressing mechanisms in MIPS ISA
  - Immediate addressing mode
    - Refers to constant operands
  - Register addressing mode
    - Refers to register operands
  - Base/Displacement addressing mode
    - Refers to memory operands
  - PC-relative addressing mode
    - Refers to the address of the conditional branch target
  - Pseudo-direct addressing mode
    - Refers to the address of the unconditional jump target and direct function call target

Manipulating strings

- C compiler stores a string as an array of characters
  - Each character is a byte and is stored as its ASCII value
  - The last character of a string is '\0' which has ASCII value zero
  - C programs involving string manipulation use lb, lbu, and sb instructions to move characters between registers and memory
- Java compiler stores a character using 16-bit Unicode values
  - Java programs involving string manipulation use lh, lhu, and sh instructions to move characters between registers and memory

Floating-point instructions

- MIPS ISA offers a separate set of floating-point instructions
  - Manipulates floating-point data
  - Floating-point registers are separate and there are 32 of them denoted $f0 to $f31, each 32 bits long in 32-bit MIPS
    - Double precision values are stored using a pair of registers e.g., the double precision value in $f_2n refers to the pair $f_2n and $f_{2n+1}
      - $f_2n contains the least significant word
    - There is no floating-point register hardwired to zero
  - All instructions are still 32 bits long
  - IEEE 754 single-precision and double-precision

Floating-point instructions

- MIPS ISA offers a separate set of floating-point instructions
  - Typical instructions are add, sub, mul, div, abs, mov, neg, compare, precision/type conversion, load, store, etc.
  - Each instruction has two flavors: .s and .d
    - add.s is a single-precision addition operation
      - add.s $f0, $f1, $f2 // $f0 ← $f1 + $f2
    - add.d is a double-precision addition operation
      - add.d $f0, $f2, $f4 // {$f1, $f0} ← {$f3, $f2} + {$f5, $f4}
  - The mul and div instructions use explicit floating-point destination register as opposed to Hi/Lo
    - mul.s $f0, $f1, $f2 // $f0 ← $f1*$f2
    - div.s $f0, $f1, $f2 // $f0 ← $f1/$f2

Floating-point instructions

- MIPS ISA offers a separate set of floating-point instructions
  - The mov instruction is needed due to absence of a hardwired zero register
    - Moves from one floating-point register to another
      - mov.s $f0, $f1 // $f0 ← $f1
      - mov.d $f0, $f2 // {$f1, $f0} ← {$f3, $f2}
  - There are instructions to move between floating-point registers and general-purpose registers
    - mfc1 and mtc1 instructions
    - mfc1 $2, $f0 // $2 ← $f0
    - mtc1 $3, $f0 // $f0 ← $3 (note the assembler syntax)
    - Needed to manipulate the bits of a floating-point number
    - Instructions having a mix of FPR and GPR operands

Floating-point instructions

- MIPS ISA offers a separate set of floating-point instructions
  - The precision/type conversion instructions can convert between various types
    - float to double
    - double to float
    - float to int
    - double to long (or long long)
    - Needed for different kinds of typecast operations
  - All floating-point instructions have a single encoding [opcode:format:ft:fs:fd:function]
    - add, sub, mul, div, abs, neg, mov, compare, type/precision conversion instructions have a common opcode (0x11) and gets distinguished by different function codes

Floating-point instructions

- MIPS ISA offers a separate set of floating-point instructions

  - Floating-point instructions do not allow immediate operands
  - Load and store instructions need to encode displacement
    - lwc1 $f0, 20($10) // $f0 ← [$10 + 20] ($f0 is ft)
    - swc1 $f0, 20($10) // [$10 + 20] ← $f0 ($f0 is ft)
    - Loading or storing a double-precision value generates two lwc1 or two swc1 instructions in 32-bit MIPS
    - lwc1 and swc1 instructions use the I-format of integer instructions where rt field specifies ft

Floating-point instructions

- Floating-point instructions do not allow immediate operands in MIPS ISA
  - This complicates translation of floating-point operations involving constants e.g., x = y + 1.5
  - One option is to prepare the IEEE 754 representation of the constant in a general-purpose register and then move it to a floating-point register using mtc1
    - Requires at least three instructions: lui, ori, mtc1
    - Need to deal with pair of registers for double-precision
      - » Requires six instructions
    - Takes away general-purpose registers for compiling floating-point operations
  - If a constant is used at many places, another option is to store it in memory and load it into a floating-point register when needed
    - Requires just one lwc1 instruction to get the constant in a floating-point register (two lwc1 for double-precision)

Translating and starting a program

- Four-step process involving four different pieces of system software: compiler, assembler, linker, loader
  - Compiler translates an HLL program into assembly language program
  - An assembly language program specifies the program in terms of instructions and pseudo-instructions
    - Pseudo-instructions are not supported by the ISA, but are presented as a convenience
      - move $4, $5 // add $4, $0, $5
      - blt $4, $5, label // slt $at, $4, $5; bne $at, $0, label
      - bgt, ble, bge
      - li $5, 0x12345 // lui $5, 0x12; ori $5, $5, 0x3456

Translating and starting a program

- Assembler translates an assembly language program into an object file
  - Cannot use pseudo-instructions any more
  - Object file contains the program text in binary (machine language program), program data, and information to place instructions in memory
  - Assembler assigns values to all labels in the program by maintaining a symbol table

Translating and starting a program

- A UNIX object file has six segments
  - Header: describes the sizes and positions of the other segments
  - Text: machine language program
  - Static data: persistent data that live throughout the life of the program
  - Relocation information: identifies instructions and data that depend on absolute addresses when the program is loaded into memory
  - Symbol table: labels whose values are not resolved yet (external references e.g., externs)
  - Debug information: how to associate instructions with HLL program statements and make data structures readable in the binary

Translating and starting a program

- Linker's job is to "stitch together" multiple object files to create one single executable binary
  - Each HLL program file is compiled and assembled independently to produce an object file
    - prog.c to prog.s to prog.o
    - All labels are resolved local to an object file by the assembler
  - Linker (also known as link editor) takes all the object files and edits the labels and addresses to make them globally consistent across all object files of the executable

Translating and starting a program

- Linker executes three steps
  - Place code and data symbolically in memory
  - Determine the data addresses and instruction labels
  - Resolve all internal and external references
- Linker uses the relocation information and the symbol table to resolve all labels and addresses
- Linker's output is an executable binary file
  - Has the same format as an object file with no unresolved label or address

Translating and starting a program

- Loader loads portions of the executable file into memory on demand, stores the main function's arguments on stack, initializes the registers and stack pointer, sets the PC to point to the first instruction of the "start-up routine"
  - The start-up routine copies the main function's arguments from stack to argument registers and executes "jal main"
  - When main function returns back to the start-up routine, the program exits
    - This is done through the "exit system call" which frees all memory allocated to the program

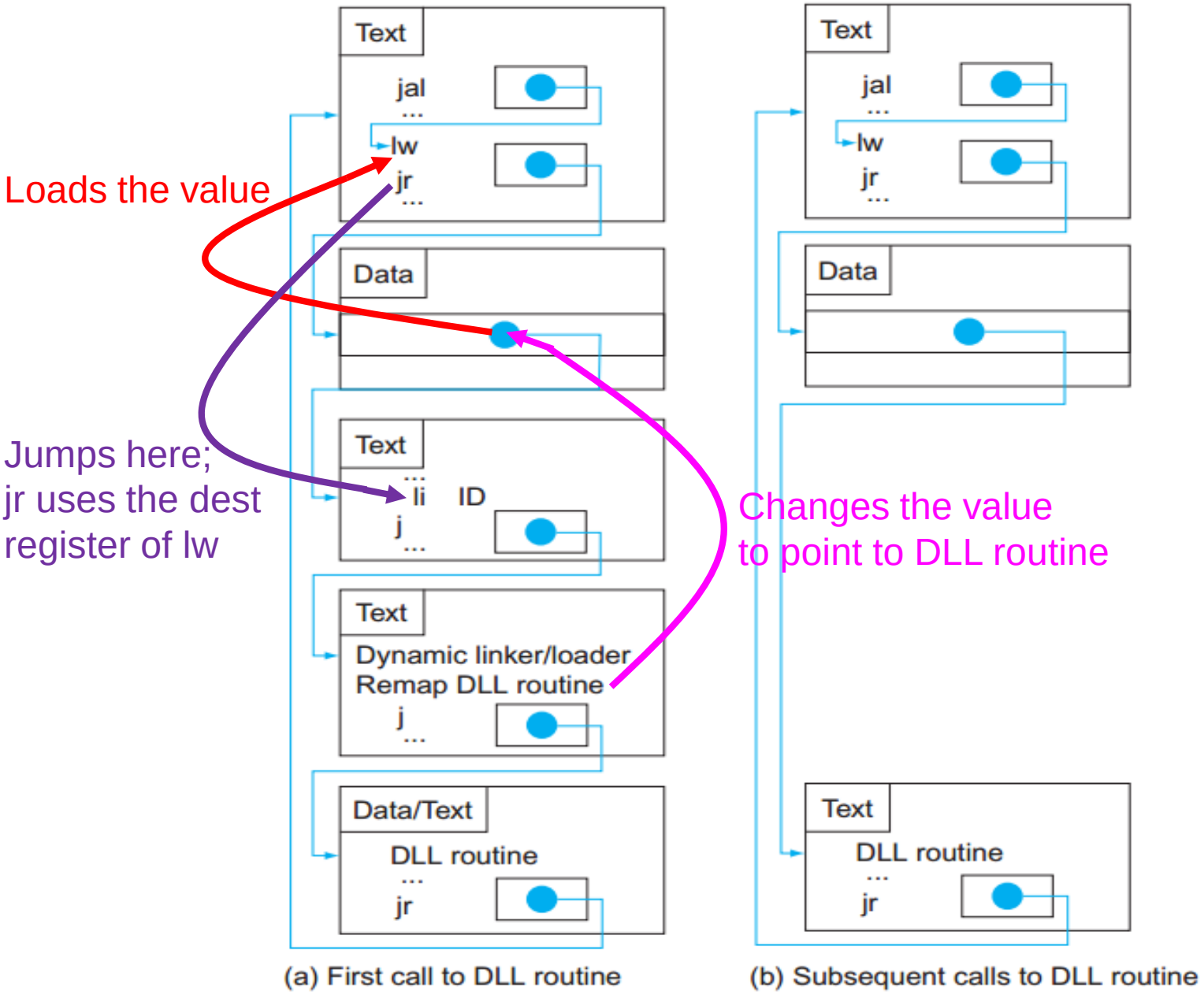
Translating and starting a program

- C programs often use library functions
  - This requires linking the program's object files with the library object file(s)
- Static linking links all required library object files with the program's object files
  - The resulting executable file can be quite large in size wasting space
  - After the executable is built, if a bug in a library gets fixed or an improved version of the library is released, the executable will continue to use the old library unless the linking procedure is repeated
  - This disadvantages are addressed through dynamic linking

Translating and starting a program

- A dynamically linked library invokes the linker only when the program calls a function belonging to the library object file
  - Linker is invoked while the program is running
  - Initial version of the dynamic linker was used to be invoked by the loader just before the program ran and the loader would link the required libraries
    - Still links all required libraries as opposed to only the library functions that are called
  - Today's dynamic linkers do lazy procedure linking where a library function is linked at the time it is called

Translating and starting a program



(a) First call to DLL routine

(b) Subsequent calls to DLL routine

Translating and starting a program

- First call to a dynamically linked library function has a very high overhead
- Subsequent calls have lower overhead, but still requires an extra indirect jump (jr)
- Dynamic linking saves space at the expense of extra instruction execution compared to static linking