MIPS32 architecture brief summary

Prof Rajeev Barua

E-A 001 -- Slide set 2

What is Machine Code vs Assembly language?

PLAKSHA UNIVERSITY

- Machine code is a sequence of instructions, each composed of 1s and 0s, specifying the code that runs on the hardware.
  - The CPU only runs machine code and nothing else! All other languages must be converted to it.

- When writing or reading machine code, humans don't find 1s and 0s readable.

- Hence Assembly language has been designed. **It is a text-based human-readable form of machine code (or object code).**
  - Converters are available to convert assembly code into object code and the opposite. (See figure from the last class).

ISAs: The format of machine code

The format of machine code is specified by the Instruction Set Architecture (ISA).

Example ISAs and their OSs:

- x86 (used in all Intel and AMD CPUs).
  - Example OSs: Windows, MacOS, Linux
- ARM (used in all mobile phones and tablets)
  - Example OSs: Android, iOS
- SPARC (used in Sun Microsystems/Oracle machines)
  - Example OSs: Solaris. Rarely used nowadays.
- IBM 360 (used in IBM mainframes)
  - Example OSs: IBM z/OS
- MIPS (used in video game consoles and embedded systems eg., networking devices)
  - Example OSs: SGI IRIX, Windows NT, Linux

Overview of the MIPS ISA

The MIPS ISA has two versions: MIPS64, and MIPS32. We will use the MIPS32 ISA as the example architecture throughout this course.

- It is considered a modern and highly efficient RISC ISA. (RISC= Reduced Instruction Set Computing)
- In contrast x86 is a CISC ISA (CISC = Complex Instruction Set Computing)
- Idea of RISC: a lot of simple instructions.
- CISC is considered inferior to RISC.
    - x86 (CISC) is more popular than MIPS (RISC) only because of binary compatibility reasons. x86 (1978) is older than MIPS (early 1985s)

Registers and memory locations

Machine code programs and their assembly counterparts do not contain the names of source code variables. Instead those variables are stored in hardware storage locations called registers and memory locations.

- Registers are collections of flip flops that are explicitly named.
  - ➢ For example, in the MIPS32 ISA, program-visible registers are $0 to $31, which names (or their alternate names) may appear in the assembly code.
  - ➢ Hardware registers are those that do not appear in assembly code, such as PC (program counter) or ALU-OUTPUT, but appear in the hardware.
- Memory locations are accessed by “memory addresses” in hardware banks of storage called main memory, implemented by SRAM or DRAM technology.
  - ➢ Memory addresses are binary numbers specifying a location in the memory to be read or written to.

Features of MIPS32 ISA

MIPS32 is a 32-bit ISA. In this course, when we refer to MIPS, we mean MIPS32.

The bitness of an ISA: ***is the width of the registers in it.***

Registers are explicitly named hardware storage locations

- MIPS has 32 registers ($0, $1, … $31). Each is 32 bits long.
    - Since the registers are 32 bits long ⇒ Bitness of MIPS32 is 32 bit.
  - Since memory addresses must be contained in registers, they must be 32 bits long at the most. (⇒ Max size of memory = 2^{32} bytes = 4 GB).
    - (GB = GigaByte. Giga = 2^{30} for memory sizes or 10^9 for everything else)
- Coincidentally, MIPS32 instructions are also all 32 bits long. (RISC ISAs have fixed length instructions).
    - However, instruction length does not necessarily equal bitness!
- Only loads and stores access data memory (like all RISC ISAs)

Prefixes of storage etc

PLAKSHA UNIVERSITY

Prefix name	Meaning in metric	Meaning in binary
Kilo	10^3	2^{10}
Mega	10^6	2^{20}
Giga	10^9	2^{30}
Tera	10^{12}	2^{40}
Peta	10^{15}	2^{50}

Most measures like hertz, grams, and meters use metric prefixes. However memory space, (like bytes) is always in binary because memory is addressed with binary numbers. So a 1KB memory is accessed by a 10 bit address, resulting in 2^10 addresses = 1KB.

Example MIPS32 instructions

PLAKSHA UNIVERSITY

- `ADD  $2, $1, $3`
  - This does $2 = $1 + $3 ($1, $2, $3 are registers)
  - The first operand (eg., $2 above) is the destination operand in most MIPS instructions
  - Remaining operand (eg., $1 and $3 above) are source operands.
- `BNE  $1, $2, _target`
  - If $1 ≠ $2, then control transfers to PC + _target, where PC is the Program Counter, which contains the address of the current instruction; else goes to next instruction.
  - _target is an 18-bit immediate (*i.e.,* constant)
  - A constant within an instruction (like `target`) is called an immediate.
- `BAL  _target`
  - Jumps to address = PC + _target, just like BNE does, but unconditionally.
  - It also stores PC + 8 into return address register, namely $31.
  - _target is a 28-bit immediate
- `LW $5, 4($6)`
  - This does $5 ← MEM($6 + 4). This means load $5 with the contents of the memory at location with address $6 + 4.

MIPS32® Instruction Set Quick Reference

RD — DESTINATION REGISTER
RS, RT — SOURCE OPERAND REGISTERS
RA — RETURN ADDRESS REGISTER (R31)
PC — PROGRAM COUNTER
ACC — 64-BIT ACCUMULATOR
LO, HI — ACCUMULATOR LOW ($ACC_{31:0}$) AND HIGH ($ACC_{63:32}$) PARTS
± — SIGNED OPERAND OR SIGN EXTENSION
∅ — UNSIGNED OPERAND OR ZERO EXTENSION
:: — CONCATENATION OF BIT FIELDS
R2 — MIPS32 RELEASE 2 INSTRUCTION
DOTTED — ASSEMBLER PSEUDO-INSTRUCTION

PLEASE REFER TO *"MIPS32 ARCHITECTURE FOR PROGRAMMERS VOLUME II: THE MIPS32 INSTRUCTION SET"* FOR COMPLETE INSTRUCTION SET INFORMATION.

ARITHMETIC OPERATIONS			
ADD	RD, RS, RT	RD = RS + RT	(OVERFLOW TRAP)
ADDI	RD, RS, CONST16	RD = RS + CONST16±	(OVERFLOW TRAP)
ADDIU	RD, RS, CONST16	RD = RS + CONST16±	
ADDU	RD, RS, RT	RD = RS + RT	
CLO	RD, RS	RD = COUNTLEADINGONES(RS)	
CLZ	RD, RS	RD = COUNTLEADINGZEROS(RS)	
LA	RD, LABEL	RD = ADDRESS(LABEL)	
LI	RD, IMM32	RD = IMM32	
LUI	RD, CONST16	RD = CONST16 << 16	
MOVE	RD, RS	RD = RS	
NEGU	RD, RS	RD = –RS	
SEB[R2]	RD, RS	$RD = RS_{7:0}^{\pm}$	
SEH[R2]	RD, RS	$RD = RS_{15:0}^{\pm}$	
SUB	RD, RS, RT	RD = RS – RT	(OVERFLOW TRAP)
SUBU	RD, RS, RT	RD = RS – RT	

SHIFT AND ROTATE OPERATIONS		
ROTR[R2]	RD, RS, BITS5	$RD = RS_{BITS5-1:0} :: RS_{31:BITS5}$
ROTRV[R2]	RD, RS, RT	$RD = RS_{RT4:0-1:0} :: RS_{31:RT4:0}$
SLL	RD, RS, SHIFT5	RD = RS << SHIFT5
SLLV	RD, RS, RT	$RD = RS << RT_{4:0}$
SRA	RD, RS, SHIFT5	$RD = RS^{\pm} >> SHIFT5$
SRAV	RD, RS, RT	$RD = RS^{\pm} >> RT_{4:0}$
SRL	RD, RS, SHIFT5	$RD = RS^{\varnothing} >> SHIFT5$
SRLV	RD, RS, RT	$RD = RS^{\varnothing} >> RT_{4:0}$

LOGICAL AND BIT-FIELD OPERATIONS		
AND	RD, RS, RT	RD = RS & RT
ANDI	RD, RS, CONST16	$RD = RS \,\&\, CONST16^{\varnothing}$
EXT[R2]	RD, RS, P, S	$RD = RS_{P+S-1:P}^{\varnothing}$
INS[R2]	RD, RS, P, S	$RD_{P+S-1:P} = RS_{S-1:0}$
NOP		NO-OP
NOR	RD, RS, RT	RD = ~(RS | RT)
NOT	RD, RS	RD = ~RS
OR	RD, RS, RT	RD = RS | RT
ORI	RD, RS, CONST16	$RD = RS \mid CONST16^{\varnothing}$
WSBH[R2]	RD, RS	$RD = RS_{23:16} :: RS_{31:24} :: RS_{7:0} :: RS_{15:8}$
XOR	RD, RS, RT	RD = RS ⊕ RT
XORI	RD, RS, CONST16	$RD = RS \oplus CONST16^{\varnothing}$

CONDITION TESTING AND CONDITIONAL MOVE OPERATIONS		
MOVN	RD, RS, RT	IF RT ≠ 0, RD = RS
MOVZ	RD, RS, RT	IF RT = 0, RD = RS
SLT	RD, RS, RT	$RD = (RS^{\pm} < RT^{\pm})\ ?\ 1 : 0$
SLTI	RD, RS, CONST16	$RD = (RS^{\pm} < CONST16^{\pm})\ ?\ 1 : 0$
SLTIU	RD, RS, CONST16	$RD = (RS^{\varnothing} < CONST16^{\varnothing})\ ?\ 1 : 0$
SLTU	RD, RS, RT	$RD = (RS^{\varnothing} < RT^{\varnothing})\ ?\ 1 : 0$

MULTIPLY AND DIVIDE OPERATIONS		
DIV	RS, RT	$LO = RS^{\pm} / RT^{\pm};\ HI = RS^{\pm} \bmod RT^{\pm}$
DIVU	RS, RT	$LO = RS^{\varnothing} / RT^{\varnothing};\ HI = RS^{\varnothing} \bmod RT^{\varnothing}$
MADD	RS, RT	$ACC \mathrel{+}= RS^{\pm} \times RT^{\pm}$
MADDU	RS, RT	$ACC \mathrel{+}= RS^{\varnothing} \times RT^{\varnothing}$
MSUB	RS, RT	$ACC \mathrel{-}= RS^{\pm} \times RT^{\pm}$
MSUBU	RS, RT	$ACC \mathrel{-}= RS^{\varnothing} \times RT^{\varnothing}$
MUL	RD, RS, RT	$RD = RS^{\pm} \times RT^{\pm}$
MULT	RS, RT	$ACC = RS^{\pm} \times RT^{\pm}$
MULTU	RS, RT	$ACC = RS^{\varnothing} \times RT^{\varnothing}$

ACCUMULATOR ACCESS OPERATIONS		
MFHI	RD	RD = HI
MFLO	RD	RD = LO
MTHI	RS	HI = RS
MTLO	RS	LO = RS

JUMPS AND BRANCHES (NOTE: ONE DELAY SLOT)		
B	OFF18	$PC \mathrel{+}= OFF18^{\pm}$
BAL	OFF18	$RA = PC + 8,\ PC \mathrel{+}= OFF18^{\pm}$
BEQ	RS, RT, OFF18	$IF\ RS = RT,\ PC \mathrel{+}= OFF18^{\pm}$
BEQZ	RS, OFF18	$IF\ RS = 0,\ PC \mathrel{+}= OFF18^{\pm}$
BGEZ	RS, OFF18	$IF\ RS \geq 0,\ PC \mathrel{+}= OFF18^{\pm}$
BGEZAL	RS, OFF18	$RA = PC + 8;\ IF\ RS \geq 0,\ PC \mathrel{+}= OFF18^{\pm}$
BGTZ	RS, OFF18	$IF\ RS > 0,\ PC \mathrel{+}= OFF18^{\pm}$
BLEZ	RS, OFF18	$IF\ RS \leq 0,\ PC \mathrel{+}= OFF18^{\pm}$
BLTZ	RS, OFF18	$IF\ RS < 0,\ PC \mathrel{+}= OFF18^{\pm}$
BLTZAL	RS, OFF18	$RA = PC + 8;\ IF\ RS < 0,\ PC \mathrel{+}= OFF18^{\pm}$
BNE	RS, RT, OFF18	$IF\ RS \neq RT,\ PC \mathrel{+}= OFF18^{\pm}$
BNEZ	RS, OFF18	$IF\ RS \neq 0,\ PC \mathrel{+}= OFF18^{\pm}$
J	ADDR28	$PC = PC_{31:28} :: ADDR28^{\varnothing}$
JAL	ADDR28	$RA = PC + 8;\ PC = PC_{31:28} :: ADDR28^{\varnothing}$
JALR	RD, RS	RD = PC + 8; PC = RS
JR	RS	PC = RS

LOAD AND STORE OPERATIONS		
LB	RD, OFF16(RS)	$RD = MEM8(RS + OFF16^{\pm})^{\pm}$
LBU	RD, OFF16(RS)	$RD = MEM8(RS + OFF16^{\pm})^{\varnothing}$
LH	RD, OFF16(RS)	$RD = MEM16(RS + OFF16^{\pm})^{\pm}$
LHU	RD, OFF16(RS)	$RD = MEM16(RS + OFF16^{\pm})^{\varnothing}$
LW	RD, OFF16(RS)	$RD = MEM32(RS + OFF16^{\pm})$
LWL	RD, OFF16(RS)	$RD = LOADWORDLEFT(RS + OFF16^{\pm})$
LWR	RD, OFF16(RS)	$RD = LOADWORDRIGHT(RS + OFF16^{\pm})$
SB	RS, OFF16(RT)	$MEM8(RT + OFF16^{\pm}) = RS_{7:0}$
SH	RS, OFF16(RT)	$MEM16(RT + OFF16^{\pm}) = RS_{15:0}$
SW	RS, OFF16(RT)	$MEM32(RT + OFF16^{\pm}) = RS$
SWL	RS, OFF16(RT)	$STOREWORDLEFT(RT + OFF16^{\pm}, RS)$
SWR	RS, OFF16(RT)	$STOREWORDRIGHT(RT + OFF16^{\pm}, RS)$
ULW	RD, OFF16(RS)	$RD = UNALIGNED\_MEM32(RS + OFF16^{\pm})$
USW	RS, OFF16(RT)	$UNALIGNED\_MEM32(RT + OFF16^{\pm}) = RS$

ATOMIC READ-MODIFY-WRITE OPERATIONS		
LL	RD, OFF16(RS)	$RD = MEM32(RS + OFF16^{\pm})$; LINK
SC	RD, OFF16(RS)	IF ATOMIC, $MEM32(RS + OFF16^{\pm}) = RD$; $RD = ATOMIC\ ?\ 1 : 0$

MD00565 Revision 01.01

PLAKSHA UNIVERSITY

Registers

0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

Default C Calling Convention (O32)

Stack Management

- The stack grows down.
  - Subtract from $sp to allocate local storage space.
  - Restore $sp by adding the same amount at function exit.
- The stack must be 8-byte aligned.
  - Modify $sp only in multiples of eight.

Function Parameters

- Every parameter smaller than 32 bits is promoted to 32 bits.
- First four parameters are passed in registers $a0–$a3.
  - 64-bit parameters are passed in register pairs:
    - Little-endian mode: $a1:$a0 or $a3:$a2.
    - Big-endian mode: $a0:$a1 or $a2:$a3.
- Every subsequent parameter is passed through the stack.
  - First 16 bytes on the stack are not used.
  - Assuming $sp was not modified at function entry:
    - The 1st stack parameter is located at 16($sp).
    - The 2nd stack parameter is located at 20($sp), etc.
  - 64-bit parameters are 8-byte aligned.

Return Values

- 32-bit and smaller values are returned in register $v0.
- 64-bit values are returned in registers $v0 and $v1:
  - Little-endian mode: $v1:$v0.
  - Big-endian mode: $v0:$v1.

MIPS32 Virtual Address Space

kseg3	0xE000.0000	0xFFFF.FFFF	Mapped	Cached
ksseg	0xC000.0000	0xDFFF.FFFF	Mapped	Cached
kseg1	0xA000.0000	0xBFFF.FFFF	Unmapped	Uncached
kseg0	0x8000.0000	0x9FFF.FFFF	Unmapped	Cached
useg	0x0000.0000	0x7FFF.FFFF	Mapped	Cached

Reading The Cycle Count Register From C

```
unsigned mips_cycle_counter_read()
{
    unsigned cc;
    asm volatile("mfc0 %0, $9" : "=r" (cc));
    return (cc << 1);
}
```

Assembly-Language Function Example

```
# int asm_max(int a, int b)
# {
#   int r = (a < b) ? b : a;
#   return r;
# }

    .text
    .set    nomacro
    .set    noreorder

    .global asm_max
    .ent    asm_max

asm_max:
    move    $v0, $a0        # r = a
    slt     $t0, $a0, $a1   # a < b ?
    jr      $ra             # return
    movn    $v0, $a1, $t0   # if yes, r = b

    .end    asm_max
```

C / Assembly-Language Function Interface

```
#include <stdio.h>

int asm_max(int a, int b);

int main()
{
    int x = asm_max(10, 100);
    int y = asm_max(200, 20);
    printf("%d %d\n", x, y);
}
```

Invoking MULT And MADD Instructions From C

```
int dp(int a[], int b[], int n)
{
    int i;
    long long acc = (long long) a[0] * b[0];
    for (i = 1; i < n; i++)
        acc += (long long) a[i] * b[i];
    return (acc >> 31);
}
```

Atomic Read-Modify-Write Example

```
atomic_inc:
    ll      $t0, 0($a0)      # load linked
    addiu   $t1, $t0, 1      # increment
    sc      $t1, 0($a0)      # store cond'l
    beqz    $t1, atomic_inc  # loop if failed
    nop
```

Accessing Unaligned Data

note: ULW and USW automatically generate appropriate code

Little-Endian Mode		Big-Endian Mode	
LWR	RD, OFF16(RS)	LWL	RD, OFF16(RS)
LWL	RD, OFF16+3(RS)	LWR	RD, OFF16+3(RS)
SWR	RD, OFF16(RS)	SWL	RD, OFF16(RS)
SWL	RD, OFF16+3(RS)	SWR	RD, OFF16+3(RS)

Accessing Unaligned Data From C

```
typedef struct
{
    int u;
} __attribute__((packed)) unaligned;

int unaligned_load(void *ptr)
{
    unaligned *uptr = (unaligned *)ptr;
    return uptr->u;
}
```

MIPS SDE-GCC Compiler Defines

__mips	MIPS ISA (= 32 for MIPS32)
__mips_isa_rev	MIPS ISA Revision (= 2 for MIPS32 R2)
__mips_dsp	DSP ASE extensions enabled
_MIPSEB	Big-endian target CPU
_MIPSEL	Little-endian target CPU
_MIPS_ARCH_*CPU*	Target CPU specified by `-march=`*CPU*
_MIPS_TUNE_*CPU*	Pipeline tuning selected by `-mtune=`*CPU*

Notes

- Many assembler pseudo-instructions and some rarely used machine instructions are omitted.
- The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters.
- The examples illustrate syntax used by GCC compilers.
- Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.

PLAKSHA UNIVERSITY

MIPS32 assembly: example1.s

```
1   # A demonstration of some simple MIPS instructions
2   # used to test QtSPIM
3
4       # Declare main as a global function
5       .globl main
6
7       # All program code is placed after the
8       # .text assembler directive
9       .text
10
11  # The label 'main' represents the starting point
12  main:
13      li $t2, 25      # Load immediate value (25)
14      lw $t3, value       # Load the word stored in value (see bottom)
15      add $t4, $t2, $t3   # Add
16      sub $t5, $t2, $t3   # Subtract
17      sw $t5, Z       #Store the answer in Z (declared at the bottom)
18
19      # Exit the program by means of a syscall.
20      # There are many syscalls - pick the desired one
21      # by placing its code in $v0. The code for exit is "10"
22      li $v0, 10 # Sets $v0 to "10" to select exit syscall
23      syscall # Exit
24
25      # All memory structures are placed after the
26      # .data assembler directive
27      .data
28
29      # The .word assembler directive reserves space
30      # in memory for a single 4-byte word (or multiple 4-byte words)
31      # and assigns that memory location an initial value
32      # (or a comma separated list of initial values)
33  value:  .word 12
34  Z:  .word 0
35
```

Does some arithmetic, and stores the result in memory.

Does not print output on the screen.

MIPS32 assembly: example2_hello_world.s

PLAKSHA UNIVERSITY

```
1   # "Hello World" in MIPS assembly
2   # From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
3
4       # All program code is placed after the
5       # .text assembler directive
6       .text
7
8       # Declare main as a global function
9       .globl  main
10
11  # The label 'main' represents the starting point
12  main:
13      # Run the print_string syscall which has code 4
14      li  $v0,4       # Code for syscall: print_string
15      la  $a0, msg    # Pointer to string (load the address of msg)
16      syscall
17      li  $v0,10      # Code for syscall: exit
18      syscall
19
20      # All memory structures are placed after the
21      # .data assembler directive
22      .data
23
24      # The .asciiz assembler directive creates
25      # an ASCII string in memory terminated by
26      # the null character. Note that strings are
27      # surrounded by double-quotes
28  msg:    .asciiz "Hello World!\n"
```

Prints the string “Hello World!” on the screen.