Assemblers, Linkers, and the SPIM Simulator

Ming-Hwa Wang, Ph.D.
COEN 210 Computer Architecture
Department of Computer Engineering
Santa Clara University

Introduction

- encoding instructions as binary numbers is natural and efficient for computers, but people read and write symbols (words) much better than long sequences of digits, therefore, a human-readable program is translated into a form that a computer can execute
- assembly language is the symbolic representation of computer's binary coding – machine language, assembly language is more readable than machine language because it uses symbols instead of bits; symbols are commonly occurring bit patterns; assembly language also uses labels (names followed by a colon) to identify and name particular memory words that hold instructions or data
- an assembler translates assembly language source file into object file (containing binary machine instructions, data, and bookkeeping information that combine several object files into a program)
- most programs consist of several files or modules; a module may contain references to subroutines and data defined in other modules and in libraries; the code in a module can't be executed when it contains unresolved references
- a linker combines a collection of object and library files into an executable file (with all external labels resolved), which a computer can run
- an assembly program's lack of control flow constructs provides few hints about the program's operation; by contrast, the C routine is both shorter and clearer since variables have mnemonic names, the loop is explicit rather than constructed with branches, and type-checking
- assembly language plays two roles: the output language of a compiler, and as a language in which to write programs; a compiler translate a high-level language program (the source file) into assembly or machine code (the target language); most programmers write only in high-level language, but assembly language is still important to write programs in which speed or size are critical or to exploit hardware features that have no analogues in high-level languages
- a compiler can produce machine language directly instead of assembly; however, it must perform the task an assembler normally handles; the trade-off is between compilation speed and compiler simplicity
- when to use assembly language
  - embedded computer: a computer that is incorporated in another device, and needs to respond rapidly and predictably to events in the outside world; an assembly language has tight control over which instructions execute, reducing a program's size also reduce the cost
  - Ada is a high-level language for writing embedded systems in DoD
  - a hybrid approach: most of the program is written in a high-level language and time-critical sections are written in assembly
  - programs typically spend most of their time executing a small fraction of the program's source code; program profiler measures where a program spends its time and can find the time-critical parts of a program
  - compilers typically are better than programmers at producing uniformly high-quality machine code across an entire program; programmers understand a program's algorithm and behavior at a deeper level than a compiler and can expend considerable effort and ingenuity improving small sections of the program
  - compilers typically compile each procedure in isolation and must follow strict conventions governing the use of registers at procedure boundary; by retaining commonly used values in registers, even across procedure boundaries, programmers can make a program run faster
  - assembly languages have the ability to exploit specialized instructions
  - a programmer's advantage over a compiler is likely to become increasing difficult to maintain as compilation techniques improve and machine's pipeline increase in complexity
  - no high-level language is available on a particular computer
- drawbacks of assembly language
  - inherently machine-specific and must be totally rewritten to run on another computer architecture
  - assembly programs are longer; the ratio of assembly to high-level language is the expansion factor; programmers write roughly the same number of lines of code per day in assembly as in high-level language; more productive in a high-level language; longer programs are more difficult to read and understand and they contain more bugs

Assemblers

- assembly language is a programming language; it provides only a few, simple types of data and control flow, and does not specify the type of value held in a variable; a programmer must apply the appropriate operations to a value, and must implement all control flow with gotos
- an assembler has 2 parts; the first step is to find memory locations with labels so the relationship between symbolic names and address is known when instructions are translated; the second step is to translate each assembly statement by combining the numeric equivalents of opcodes, registers specifiers, and labels into a legal instruction
  - forward reference: labels maybe used before they are defined, which forces an assembler to translate a program in two steps
  - backpatching: speed assembly because the assembler only reads the input once, but it requires an assembler to hold the entire binary representation of the program in memory so instruction can be backpatched (limit the size of programs that can be assembled)

- unix object file format:
  - object file header: describes the size and position of the other pieces of the file
  - text segment: contains the machine code for routines in the source file; these routines maybe unexecutable because of unresolved references
  - data segment: contains a binary representation of the data in the source file; the data also maybe incomplete because of unresolved references to labels in other file
  - relocation information: identifies instruction and data words that depend on absolute address; these reference must change if portions of the program are moved in memory; procedures and data from a file are stored in a contiguous of memory, but the assembler does not know where this memory will be located; for convenience, assemblers assume each file starts at the same address (location 0) with the expectation that the linker will relocate the code and data when they are assigned locations in memory; instruction that use PC-relative addressing (e.g., branches), need not be relocated
  - symbol table: associates addresses with external labels in the source file and lists unresolved references
  - debugging information: contains a concise description of the way in which the program was compiled, so a debugger can find which instruction address correspond to lines in a source file and print the data structures in readable form
- additional facilities or convenience features: help make assembler programs short and easier to write
  - assembler directives (names that begin with a period) tell the assembler how to translate a program but do not produce machine instructions
    - **.text** indicates that succeeding lines contain instructions
    - **.data** indicate that they contain data
    - **.align** *n* indicates that the items on the succeeding lines should be aligned on a 2^n byte boundary
    - **.global** *main* declares that *main* is a global or external symbol that should be visible to code stored in other file; labels are local by default, so the compiler needs not produce unique names in every file
    - **.asciiz** stores null-terminated string in memory; data layout directives allow a programmer to describe data in a more concise and natural manner than its binary representation that the assembler will translate it into binary
  - macros enable a programmer to extend the assembly language by defining new operations; macros are a pattern-matching and replacement facility that provide a simple mechanism to name a frequently used sequence of instructions; instead of repeatedly typing the same instructions every time they are used, a programmer invokes the macro and the assembler replaces the macro call with the corresponding sequence of instructions; a macro call is replaced by the macro's body when the program is assembled
    - **.macro**
    - **.end_macro**
    - **$arg**: formal parameter: names the argument to the macro; when the macro is expanded, the argument from a call is substituted for the formal parameter throughout the macro's body; then the assembler replaces the call with the macro's newly expanded body
  - pseudoinstructions are instruction provided by an assembler but not implemented in hardware; they make assembly language programming easier without complicating the hardware; many pseudoinstructions could also be simulated with macros, but MIPS can generate better code by using a dedicated register **($at**) and is able to optimize the generated code
  - conditional assembly: permits a programmer to include or exclude groups of instructions when a program is assembled (e.g., cpp, m4)

Linkers

- separate compilation permits a program to be split into pieces that are stored in different files; each file contains a logically related collection of subroutines and data structures that from a module in a large program; a file can be compiled and assembled independently, so changes to one module do not require recompiling the entire program
- the linker
  - searches the program libraries to find library routines used by the program, and incorporates library routines into the program text segment
  - determines the memory location that code from each module will occupy and relocates its instructions by adjusting absolute references
  - resolves references among files
- the linker produces an executable file that can be run on a computer; typically, this file has the same format as an object file, except that it contains no unresolved reference or relocation information

Loading

- the operating system kernel brings a program into memory and starts it running
  1. reads the executable file's header to determine the size of the text and data segment
  2. creates a new address space for the program, along with a stack segment
  3. copies instructions and data from the executable file into the new address space
  4. copies arguments passed to the program onto the stack
  5. initializes the machine registers; the stack pointer must be assigned the address of the first free stack location

6. jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's **main** routine; if the **main** routine returns, the start-up routine terminates the program with the exit system call

Memory Usage

- assembly programmers use MIPS hardware by following a set of conventions or agreements (not required by hardware) to make different programmers working together and to make effective use of the hardware
- divide the memory into 3 parts:
  1. the text segment holds the program's instruction, starting at address 0x40_0000 (near the bottom of the address space)
  2. the data segment
     - static data starts at address 0x1000_0000, contains objects whose size is known to the compiler and whose lifetime is the program's entire execution (e.g., global variables)
     - dynamic data starts immediately above static data; use the **malloc** library routine to expand the dynamic data as needed; **malloc** expands the dynamic area with the **sbrk** system call, which causes the OS to add more pages to the program's virtual address space
  3. the program stack segment starts at address 0x7fff_ffff; as the program pushes values on the stack, the OS expands the stack segment down, towards the data segment; the 2 dynamically expandable segments are as far apart as possible, and they can grow to use a program's entire address space
- to avoid repeating the **lui** instruction at every load and store, MIPS system typically dedicate a register (**$gp**) as a global pointer to the static data segment; this register contains address 0x1000_8000, so load and store instruction can use their signed 16-bit offset fields to access the first 64KB of the data segment; **$gp** makes addressing locations 0x1000_0000 to 0x1001_0000 faster than other heap locations

Procedure Call Convention

- register use or procedure call conventions
  - MIPS contains 32 general-purpose registers; register **$0** always contains the hardware value 0
  - registers **$at** (1), **$k0** (26) and **$k1** (27) are reserved for the assembler and OS
  - registers **$a0**-**$a3** (4-7) are used to pass the first 4 arguments to routines (remaining arguments are passed on the stack); registers **$v0** (2) and **$v1** (3) are used to return values from functions
  - registers **$t0**-**$t9** (8-15, 24-25) are caller-saved registers that are used to hold temporary quantities that need not be preserved across calls
  - registers **$s0**-**$s7** (16-23) are callee-saved registers that are used to hold long-lived values that should be preserved across calls
  - register **$gp** (28) is a global pointer that points to the middle of a 64K block of memory in the static data segment
  - registers **$sp** (29) is the stack pointer, which points to the last location on the stack; registers **$fp** (30) is the frame pointer; the **jal** instruction writes register **$ra** (31), the return address from a procedure call
- use the names instead of register numbers; names of these registers reflect the registers' intended uses in the procedure call convention
- programmers who write in a high-level language never see the details of how one procedure (the caller) invokes another procedure (the callee) because the compiler takes care of this low-level bookkeeping; assembly language programmers must explicitly implement every procedure call and return
- a procedure call or stack frame holds values passed to a procedure as arguments, saves registers that a procedure may modify, and provides space for variables local to a procedure; in most programming languages, procedure calls and returns follow a strict LIFO order; the stack grows down from higher memory address; the executing procedure uses the frame pointer to quickly access values in its stack frame
- MIPS calling convention
  - the caller 1) passes arguments, 2) saves caller-saved registers, and 3) executes a **jal** instruction (jump to the callee's first instruction and saves the return address in register **$ra**)
  - the callee 1) allocates memory for the frame by subtracting the frame's size from the stack pointer, 2) saves callee-saved register in the frame (**$s0**-**$s7**, **$fp**, and **$ra** if the callee itself makes a call), and 3) establish the frame pointer by adding the stack frame's size minus four to **$sp** and storing the sum in register **$fp**
  - callee executes the body of the subroutine
  - the callee 1) places the return value in register **$v0** if it returns a value, 2) restores all callee-saved registers, 3) pops the stack frame by adding the frame size to **$sp**, and 4) return by jumping to the address in register **$ra**
- callee-saved registers are better used to hold long-lived values, these registers are only saved during a procedure call if the callee expects to use the register; caller-saved registers are better used to hold short-lived quantities that do not persist across a call
- a programming language that does not permit recursive procedures need not allocate frames on a stack, each procedure's frame may be statically allocated since only one invocation of a procedure can be active at a time; on load-store architecture (e.g., MIPS), stack frame may be just as fast because a frame pointer register points directly to the active stack frame, which permit a single load or store instruction to access values in the frame
- the difference between the MIPS compiler and the gcc compiler: the MIPS compiler usually does not use a frame pointer, so this register is available as another callee-saved register (**$s8**); this saves a couple of instructions in the procedure call and return sequence, but complicates code

generation because a procedure must access it stack frame with **$sp**, whose value can change during a procedure's execution if values are pushed on the stack

Exceptions and Interrupts

- exceptions caused by error during an instruction's execution, and interrupts caused by I/O device
- exception and interrupt handling: in MIPS processors, coprocessor 0 records the information in the following 4 registers (accessed by the **lwc0**, **mfc0**, **mtc0**, and **swc0**):
  - BadVAddr (register #8) for the memory address
  - Status (register #12) for interrupt mask and enable bits
    - the interrupt mask field (bits 15:8) contains a bit for each of the 5 hardware and 3 software possible interrupts; a bit that is 1 allows interrupts at that level; a bit that is 0 disables interrupts at that level
    - 3-deep stack (current stack uses bits 1:0, previous 3:2, and old 5:4) for the kernel(0)/user(1) and interrupt enable(1)/disable(0) bits; when an interrupt occurs, these 6 bits are shifted left by 2 bits
  - Cause (register #13) for exception type and pending interrupt bits:
    - 5 pending interrupt bits correspond to the 5 interrupt levels, and 5 exception code bits describes the cause of an exception with the following codes:
      - INT (0) for external interrupt
      - ADDRL (4) for address error exception (load or fetch)
      - ADDRS (5) for address error exception (store)
      - IBS (6) for bus error on instruction fetch
      - DBUS (7) for bus error on data load or store
      - SYSCALL (8) for syscall exception
      - BKPT (9) for breakpoint exception
      - RI (10) reserved instruction exception
      - OVF (12) arithmetic overflow exception
  - EPC (register #14) for instruction address
- interrupt handler (at address 0x80000080 in kernel address space): examines the exception's cause and jumps to an appropriate point in the OS, either by terminating the process or by performing some action and resumes the interrupted process
  - interrupt handler saves **$a0** and **$a1** in two memory locations (save0 and save1), moves the Cause and EPC registers into $k0 and $k1, tests if the exception was caused by an interrupt (if so, the interrupt is ignored, otherwise, call print_excp to print a warning message), restores $a0 and $a1, and executes **rfe** (which restores the previous interrupt mask and kernel/user bits in the Status register)

Input and Output

- SPIM simulates one I/O device: a memory-mapped terminal (xpsim) by using the –mapped_io flag; the control-C causes SPIM to stop and return to command mode
- the terminal consists of 2 independent units:
  - the receiver reads characters from the keyboard
  - the transmitter writes characters to the display
- a program controls the terminal with 4 memory-mapped device registers:
  - the Receiver Control register (at address 0xffff0000): bit 0 is a read-only "ready" bit (1 for a character has arrived from the keyboard but not read yet), and bit 1 is the keyboard "interrupt enable" (if it is 1, the terminal requests an interrupt at level 0 whenever the ready bit is 1)
  - the Receiver Data register (0xffff0004): the read-only low-order 8 bits is the typed character from keyboard, and reading the Receiver Data register resets the ready bit in the Receiver Control register
  - the Transmitter Control register (0xffff0008): bit 0 is a read-only "ready" bit, and bit 1 is "interrupt enable" (if it is 1, the terminal requests an interrupt on level 1 whenever the ready bit is 1)
  - the Transmitter Data register (0xffff000c): when a value is written in this location, its low-order 8 bits are sent to the console, and reset the ready bit in the Transmitter Control register for a while and then become 1

SPIM

- PSIM (MIPS spelled backwards) is a simulator that executes MIPS programs (either assembly or executable), it contains a debugger and provides a few OS-like services; SPIM is much slower than a real computer, but at low cost and wide availability
- simulators are a useful tool in studying computers and the programs that run on them; because they are software, they can be easily modified at add new instructions
- simulation of a virtual machine
  - RISC computers are difficult to program directly because of delayed branches, delayed loads, and restricted address modes; MIPS wisely chose to hide this complexity by having it assembler implement a virtual machine, which appears to have nondelayed branches and loads and a richer instructions set than the actual hardware; the assembler reorganizes or rearranges instructions to fill the delay slots; the virtual computer provides pseudoinstructions, the assembler translates them into equivalent sequences of actual, machine language; SPIM simulates the richer virtual machine, it can also simulate the bare machine
- getting started with SPIM
  - spim: command-line-driven program
    - exit

- read “file”
- load “file” synonym for read
- execute “a.out”
- run <addr>
- step <N>
- continue
- print $N print register N
- print $fN print floating point register N
- print addr print the content of memory at addr
- print_sym print the names and addresses
- reinitialize
- breakpoint addr
- delete addr
- list
- .
- <nl>
- ? print a help page

- xspim: the X-window program; it divides the window into 5 panes:
  - register display, which is updated whenever your program stops running
  - the control buttons
    - quit
    - load
    - run: use control-C to stop the program, continue or abort
    - step
    - clear
    - set value
    - print
    - breakpoint: add, delete, list breakpoints
    - help
    - terminal
    - mode
  - the text segments
    - the text segments consists of the hexadecimal memory address of the instruction, the instruction’s numerical encoding in hex, the instruction’s mnemonic description, the operands, the ‘;’ for comments, the line number, and the instruction (if it is not a pseudoinstruction)
  - the data and stack segments
  - the SPIM message
- command-line options

  -bare simulate a bare MIPS machine
  -asm simulate the virtual MIPS machine (the default)
  -pseudo contain pseudoinstruction (the default)
  -nopseudo
  -notrap do not load the standard exception handler & start-up code
  -trap (the default)
  -noquiet print a message when an exception occurs (the default)
  -quiet
  -nomapped_io (the default)
  -mapped_io
  -file load and execute the assembly code in the file
  -execute load & execute the code in the MIPS executable file a.out
  -s<seg>size set segment (text, data, stack, ktext, kdata) size
  -l<seg>size sets the limit on segment size
  - PCspim for Windows
- SPIM does not simulate caches or memory latency, nor instruction delays; SPIM does not reorganize instructions to fill delay slots; MIPS processors can operate with either big-endian or little-endian byte order, and SPIM byte order is the same as the byte order of the underlying machine
- system calls through the syscall instruction
  - to request a service, load the system call code into register **$v0** and arguments into registers **$a0-$a3** (or **$f12** for floating-point values); system calls that return values in **$v0** or **$f0**
  - system call codes for system services: print_int (1), print_float (2), print_double (3), print_string (4), read_int (5), read_float (6), read_double (7), read_string (8), sbrk (9), exit (10)

MIPS R2000 Assembly Language

- 3 trends are reducing the need to write programs in assembly languages
  - improvement of compiler in execution speed
  - high-level languages are machine independent
  - increasing complex applications need modularity and semantic checking features in high-level languages
- addressing modes
  - MIPS is a load-store architecture, only load and store instructions access memory; computation instructions operate only on values in registers
  - the bare machine provides only one memory-addressing mode: imm(rx)
  - the virtual machines provides (rx), imm, imm(rx), label, label ± imm, label ± imm(rx)
- assembler syntax
  - comments begin with a sharp sign (#)
  - opcodes are reserved words
  - a label is followed by a colon
  - numbers are base 10 by default, 0x for hexdecimals
  - use C strings
- MIPS assembler directives

- **.align** n
- **.ascii** str (no null-termination)
- **.asciiz** str (null-termination)
- **.byte** b1, .., bn
- **.data** <addr>
- **.double** d1, .., dn
- **.extern** sym size
- **.float** f1, .., fn
- **.global** sym
- **.half** h1, .., hn
- **.kdata** <addr>
- **.ktext** <addr>
- **.set** noat
- **.set** at
- **.space** n
- **.text** <addr>
- **.word** w1, .., wn

- arithmetic and logical instructions (pseudo codes using dest, src)
  - absolute value **abs** rdest, rsrc
  - addition with overflow **add** rd, rs, rt
  - addition without overflow **addu** rd, rs, rt
  - addition immediate with overflow **addi** rt, rs, imm
  - addition immediate without overflow **addu** rt, rs, imm
  - and **and** rd, rs, rt
  - and immediate **andi** rd, rs, rt
  - divide with overflow **div** rdest, rsrc1, rsrc2 or **div** rs, rt
  - divide without overflow **divu** rdest, rsrc1, src2 or **divu** rs, rt
  - multiply **mult** rs, rt
  - unsigned multiply **multu** rs, rt
  - multiply without overflow **mul** rdest, rsrc1, rsrc2
  - multiply with overflow **mulo** rdest, rsrc1, rsrc2
  - unsigned multiply with overflow **mulou** rdest, rsrc1, rsrc2
  - negate value with overflow **neg** rdest, rsrc
  - negate value without overflow **negu** rdest, rsrc
  - nor **nor** rd, rs, rt
  - not **not** rdest, rsrc
  - or **or** rd, rs, rt
  - or immediate **ori** rt, rs, imm
  - remainder **rem** rdest, rsrc1, rsrc2
  - unsigned remainder **remu** rdest, rsrc1, rsrc2
  - shift left logical **sll** rd, rt, shamt
  - shift left logical variable **sllv** rd, rt, rs
  - shift right arithmetic **sra** rd, rt, shamt
  - shift right arithmetic variable **srav** rd, rt, rs
  - shift right logical **srl** rd, rt, shamt
  - shift right logical variable **srlv** rd, rt, rs
  - rotate left **rol** rdest, rsrc1, rsrc2
  - rotate right **ror** rdest, rsrc1, rsrc2
  - subtract with overflow **sub** rd, rs, rt
  - subtract without overflow **subu** rd, rs, rt
  - exclusive or **xor** rd, rs, rt
  - xor immediate **xori** rd, rs, rt
- constant-manipulating instructions
  - load upper immediate **lui** rt, imm
  - load immediate **li** rdest, imm
- comparison instructions
  - set less than **slt** rd, rs, rt
  - set less than unsigned **sltu** rd, rs, rt
  - set less than immediate **slti** rt, rs, imm
  - set less than unsigned immediate **sltiu** rt, rs, imm
  - set equal **seq** rdest, rsrc1, rsrc2
  - set greater than equal **sge** rdest, rsrc1, rsrc2
  - set greater than equal unsigned **sgeu** rdest, rsrc1, rsrc2
  - set greater than **sgt** rdest, rsrc1, rsrc2
  - set greater than unsigned **sgtu** rdest, rsrc1, rsrc2
  - set less than equal **sle** rdest, rsrc1, rsrc2
  - set less than equal unsigned **sleu** rdest rsrc1, rsrc2
  - set not equal **sne** rdest, rsrc1, rsrc2
- branch instructions
  - branch **b** label
  - branch coprocessor z true **bczt** label
  - branch coprocessor z false **bczf** label
  - branch on equal **beq** rs, rt, label
  - branch on greater than equal zero **bgez** rs, label
  - branch on greater than equal zero and link **bgezal** rs, label
  - branch on greater than zero **bgtz** rs, label
  - branch on less than equal zero **blez** rs, label
  - branch on less than and link **bltzal** rs, label
  - branch on less than zero **bltz** rs, label
  - branch on not equal **bne** rs, rt, label
  - branch on equal zero **beqz** rsrc, label
  - branch on greater than equal **bge** rsrc1, rsrc2, label
  - branch on greater than equal unsigned **bgeu** rsrc1, rsrc2, label
  - branch on greater than **bgt** rsrc1, rsrc2, label
  - branch on greater than unsigned **bgtu** rsrc1, rsrc2, label
  - branch on less than equal **ble** rsrc1, rsrc2, label
  - branch on less than equal unsigned **bleu** rsrc1, rsrc2, label
  - branch on less than **blt** rsrc1, rsrc2, label

- branch on less than unsigned **bltu** rsrc1, rsrc2, label
- branch on not equal zero **bnez** rsrc, label

- jump instructions
  - jump **j** target
  - jump and link **jal** target
  - jump and link register **jalr** rs, rd
  - jump register **jr** rs
- load instructions
  - load address **la** rdest, address
  - load byte **lb** rt, address
  - load unsigned byte **lbu** rt, address
  - load halfword **lh** rt, address
  - load unsigned halfword **lhu** rt, address
  - load word **lw** rt, address
  - load word coprocessor **lwcz** rt, address
  - load word left **lwl** rt, address
  - load word right **lwr** rt, address
  - load doubleword **ld** rdest, address
  - unaligned load halfword **ulh** rdest, address
  - unaligned load halfword unsigned **ulhu** rdest, address
  - unaligned load word **ulw** rdest, address
- store instructions
  - store byte **sb** rt, address
  - store halfword **sh** rt, address
  - store word **sw** rt, address
  - store word coprocessor **swcz**, rt, address
  - store word left **swl** rt, address
  - store word right **swr** rt, address
  - store doubleword **sd** rsrc, address
  - unaligned store halfword **ush** rsrc, address
  - unaligned store word **usw** rsrc, address
- data move instructions
  - move **move** rdest, rsrc
  - move from hi **mfhi** rd
  - move from lo **mflo** rd
  - move to hi **mthi** rs
  - move to lo **mtlo** rs
  - move from coprocessor z **mfcz** rt, rd
  - move double from coprocessor 1 **mfc1**.d rdest frsrc1
  - move to coprocessor z **mtcz** rd, rt
- floating-point instructions
  - floating-point absolute value double **abs.d** fd, fs
  - floating-point absolute value single **abs.s** fd, fs
  - floating-point addition double **add.d** fd, fs, ft
  - floating-point addition single **add.s** fd, fs, ft
  - compare equal double **c.eq.d** fs, ft
  - compare equal single **c.eq.s** fs, ft
  - compare less than equal double **c.le.d** fs, ft
  - compare less than equal single **c.le.s** fs, ft
  - compare less than double **c.lt.d** fs, ft
  - compare less than single **c.lt.s** fs, ft
  - convert single to double **cvt.d.s** fd, fs
  - convert integer to double **cvt.d.w** fd, fs
  - convert double to single **cvt.s.d** fd, fs
  - convert integer to single **cvt.s.w** fd, fs
  - convert double to integer **cvt.w.d** fd. fs
  - convert single to integer **cvt.w.s** fs, fs
  - floating-point divide double **div.d** fd, fs, ft
  - floating-point dived single **div.s** fd, fs, ft
  - load floating-point double **l.d** fdest, address
  - load floating-point dingle **l.s** fdest, address
  - move floating-point double **mov.d** fd, fs
  - move floating-point single **mov.s** fd, fs
  - floating-point multiply double **mul.d** fd, fs, ft
  - floating-point multiply single **mul.s** fd, fs, ft
  - negate double **neg.d** fd, fs
  - negate single **neg.s** fd, fs
  - store floating-point double **s.d** fdest, address
  - store floating-point single **s.s** fdest, address
  - floating-point subtract double **sub.d** fd, fs, ft
  - floating-point subtract single **sub.s** fd, fs, ft
- exception and interrupt instructions
  - return from exception **rfe**
  - system call **syscall**
  - break **break** code
  - no operation **nop**