This project is composed of 5 parts. The overall goal (part 5) is to write a simulator for a subset of the MIPS R4000 instruction set that includes a virtual memory simulator and a loader. This loader will accept Unix System V standard “ELF” (Execution and Linking Format) executable files that are produced by the GNU C compiler and linker software built for MIPS. Three of the first 4 parts are separately testable demonstrations of components that will go into part 5.

Part 1 “cpu”: Simulator for the MIPS R4000 CPU that includes registers, a text dialog interface for memory, an elementary memory system that resembles the hardware support for bootstrap loading, some reports of the CPU simulation state and operation, and an elementary pipeline simulation to support delayed branches. It should also simulate a simple memory mapped output device so test programs can do output.

Part 2 Learn to use gcc, as and ld, to produce MIPS “ELF” executable files (with objdump to view their outputs and objcopy to produce boot image files).

Part 3 “mapper”: Virtual memory simulator that maintains a memory mapping table, allocates C++ free store (dynamic) memory to simulate physical memory, and performs the memory access operations requested by the “cpu”.

Part 4 “elftool”: Open, read and analyze components of an executable file in System 5 “ELF” format.

Part 5 “simmips”: Integrate the components of parts 1, 3 and 4, create a stack segment, and initialize any other needed process state to build a simulator for a simplified “abstract machine” that executes the process initialized from an executable file.

For each part submission, a Makefile must be supplied so that the command gmake programe will build an executable named programe from all (and only) the user supplied program and header files in the current directory. Here, programe is cpu, mapper, elftool or simmips depending on the Part.

Late submissions are welcome and will count according to the following formula: $G = \max(0, OG \times (7 - ND)/7)$ where $OG$ is the grade you would have gotten had the submission been on time, $ND$ is the number of days past 7:30 PM Wednesday, and $G$ is the grade that your project will get, including the late penalty.

Part 1 “cpu”

1. Recognize and record (in global variables, or in a global or passed “options structure”) whether each of the following options are given in the command line: --state-trace,
--decode-trace. (See also item (8).) For simplicity, ignore unrecognized or repeated
options. (You can add your own options to activate various debugging messages if you
wish.)

2. Set up and initialize data structures to simulate the 31-32 general purpose integer registers
($0$ to $31$), the program counter (PC), fetching PC (fetchPC) to implement delayed
control transfer instructions, instruction register (IR) to hold the instruction to execute
now, and the HI and LO registers (used with the MULT and MULCU instructions).

3. When the --state-trace option was given, print the contents of the above state variables
immediately before the next instruction execution simulation will begin and immediately
before the simulation halts. Thus, the initial state plus the state after each instruction
execution will be displayed. Macros for printing the contents of these (simulated) registers
are given in \texttt{MP.h}

4. Initialize the CPU state as follows: Initialize the registers (somewhat after (\texttt{mABI 3-31} ))
so $S2 = 0$ and $S31 = 0$. The other general purpose registers (except for $S0$, which always
contains 0) have undefined values. This includes the stack pointer register $SP = S9$. For
this project, make those values 0xdeadbeef. Initialize $PC = -1$ and $\text{fetchPC} = 0$.

5. After the CPU registers are initialized, the program runs the main simulation loop. See
the Simulation Loop section below.

6. (Memory access operations) Write functions to simulate the memory access operations
needed for instruction fetching and for data processing of Byte (8 bit) and Word (32 bit)
integer data. All word accesses must be aligned.

Many of the memory operations on RISC style machines such as MIPS apply only to
aligned 32 bit (i.e. 4 byte) words or to aligned 16 bit halfwords. This restriction can allow
the computer hardware to run faster because the logic gates that would be needed to shift
any unaligned data could be omitted from the critical timing paths. An aligned word
begins at an address that is a multiple of 4. Multiples of 4 have the 2 lower bits zero. The
word is comprised of the bytes at addresses $4n$, $4n+1$, $4n+2$ and $4n+3$. Similarly, half
words are aligned at even addresses (multiples of 2). The bits and bytes of words and half
words are operated in bigendian mode. That means that when the word is interpreted as
a number, the byte at address $4n$ has the most significant bits and the byte at address
$4n+3$ has the least significant bits.

7. If the \texttt{--boot-record bootfile} option was NOT given, implement the simulated mem-
ory access operations, for both instructions and data, as follows: Here, “0xaaaaaaa”
signifies a simulated address in hexadecimal notation (for your simulator to print) and
“0xddddddd” or “0xdd” signifies simulated data or instructions in hexadecimal notation
for your simulator to print when it requests a memory write or for you to type in when
the simulator requests a memory read. Each simulator memory request is terminated by
a newline.

\begin{itemize}
\item The CPU requests to read a data word (32 bits). Simulator prints
\texttt{RDW 0xaaaaaaa}?
\end{itemize}
You type$^1$:
0xdddddddd

- The CPU requests to read an instruction data word (32 bits). Simulator prints
  RIW 0xffffffff?
  You type:
  0xdddddddd
  which is delivered to the simulator.

- The CPU requests to read a data byte (8 bits). Simulator prints
  RDB 0xffffffff?
  You type:
  0xdd

- The CPU requests to write a data word (32 bits). Simulator prints
  WDW 0xffffffff 0xdddddddd
  You type:
  OK

- The CPU requests to write a data byte (8 bits). Simulator prints
  WDB 0xffffffff 0xdd
  You type:
  OK

8. If the option --boot-record filename was given, try to open for reading the named file. If the opening or file reading fails, print the relevant message in Project2/Messages/MFileErrors.h Read 512 bytes into (a properly allocated) array of long unsigned ints. Instead of interacting with the user to simulate memory access operations, use this array to simulate memory. The MIPS simulated address range simulated by this array is 0 to 511 (base ten), i.e., 0x0 to 0xff.

For information about reading “unformatted” files, begin with Deitel and Deitel’s section 11.5 on Unformatted I/O. Then study the documentation for the ifstream constructors, read, seekg, and gcount. See

- man ifstream and man istream on eve or veggies.

- Emacs info pages:
  Libg++/IOStream/Files and Strings/Files ifstream
  Libg++/IOStream/Streams/Istream/String Input read
  Libg++/IOStream/Streams/Istream/Input Position seekg
  (0r use links from them in Libg++/IOStream/Index)

- Use the html documentation linked from Textbooks and other Reading Resources on the CSI402 home page.

Read about the Unix library functions open, read, lseek and write in say Topics in C. Contrast them with the corresponding iostream facilities.

$^1$Enable user supplied input radix (base) by cin.setf(0,ios::basefield)
9. (Simple Output Device) The CSI402 MIPS machine has a bus into which is plugged a very simple output device that illustrates how memory mapped i/o devices are controlled by programs. Following (MipsProc p.75-76, pdf.105-106), the CPU will run in (simulated) 32 bit kernel mode, so that machine instructions can access the (simulated) hardware bus as well as use interactive or bootstrap memory (and virtual memory when installed in Part 5).

See Project2/Arch/sod.h for the specifications for our Simple Output Device (SOD). If you implement SOD, your test programs can write output to your terminal! Activate the SOD only for programs loaded from boot or ELF files.

When any instruction stores a word to address 0xa00f000, the low byte of that word is printed (actually, to written to stdout). Location 0xa00f000 cannot be read (loaded) by the program. This will cause a bus error. However, at address 0xa00f004 is a control register. It can be read and written by the cpu program. When the low bit (value 0x1, defined by macro SODTestModeBit in sod.h) is set(1), the character should be converted to an integer and printed with macro MSODTestOutput. When the low bit is clear (0), the character should just be printed as a character.

Simulation Loop

A non-pipelined CPU runs a simple instruction fetch/execute loop (after initialization): Fetch the instruction from memory at the address currently in the PC, execute that instruction (which makes register and/or memory state changes), check for error or exception conditions, and produce a new value in the PC (either by incrementing the PC by the length of the last instruction or setting the PC to the target address from a jump or taken branch).

The classic 5 stage pipeline for RISC CPUs is described in Patterson and Hennessy’s book. The MIPS R4400 uses an 8 stage pipeline described in (MipsProc p.48, section 3.3). The pipeline permits the CPU hardware to perform some of the work for 8 successive instructions at the same time under ideal conditions. However, when a control transfer instruction (branch or jump) is taken, the work done in behalf of instructions that immediately follow the control transfer instruction and enter the pipeline before the CPU can resolve the control transfer target results in some wasted CPU cycles. To reduce the number of cycles that are wasted, the instruction set architecture specifies that control transfers are delayed.

Delayed control transfer means that if the control transfer instruction is located at address PC, the instruction immediately following in memory, at address PC+4, is executed even when the control transfer is taken. Let DESTPC be the control transfer target address. When the control transfer is taken, the instructions actually executed are those located at addresses PC, PC+4 and then DESTPC.

You can simulate a simple 2 stage pipeline that illustrates the benefit of delayed control transfers and produces the behavior of the MIPS CPU. The two stage organization means that the execution (data processing) of the current instruction and the fetching (from memory) of the NEXT instruction are both done at the same time. Therefore, if the current instruction at PC is a control transfer, the instruction at PC+4 is fetched while the control
transfer conditions and target address are calculated. In the next cycle, the execution of the instruction at PC+4 is completed while the instruction at the target address is fetched.

Your simulator cannot actually perform the two stages at the same time since C/C++ has no way to express parallelism\(^2\). The simulator should simulate parallelism with two separate blocks of code, independent of one another, which you will put one after the other.

Let’s call one execution of the body of the simulation loop a *cycle*. The simulation loop control will use three variables:

1. **fetchPC** is the address of the instruction to be fetched (for the NEXT cycle.)
2. **PC** is the address of the instruction to be executed in THIS cycle.
3. **IR** (Instruction Register) will hold the instruction that was fetched in the previous cycle and which will be executed in this cycle.

Therefore, **PC** and **IR** satisfy the invariant \( IR = Mem[PC] \).

Here is the simulation loop outline:

1. If the **--trace-state** option was given, print the current **PC**, **IR**, **fetchPC** and all general register contents using the macros in *Project2/Messages/MP.h*

2. (execute the current instruction and fetch the next instruction in parallel) In “parallel,” do the two steps:
   - Interpret the instruction in **IR**. Record in variables of your design whether it is a taken control transfer and what the target address is.
     
     If the **--decode-trace** option was given, print the relevant one of three reports on the instruction, its decoding and CAPITALIZED name using macros in *Project2/Messages/MP.h*
   - Fetch the instruction at the simulated address given by the current value of **fetchPC**. Check if the fetch operation was a legal memory access. (Use your own temporary variables to record whether the memory access was successful and, if so, the fetched instruction.)

3. (check for errors and update the loop control variables) If the instruction execution or the fetch had errors, report the errors and stop the simulation loop. If the executed instruction was a **SYSCALL**, stop the loop. Print the relevant “stop reason” from *Project2/Messages/MP.h* Print the final state trace message if the option **--state-trace** was given (also from **MP.h**).

   Otherwise, do the steps below sequentially:

   (a) Copy the fetched instruction into **IR**.

\(^2\)eve.albany.edu is a multiprocessor machine but the subject of how to program it with threads to do computations for one process in parallel is beyond the scope of this project.
(b) Copy fetchPC into PC. (PC=fetchPC)
(c) Compute and store into fetchPC either PC+4, PC+Displacement, or AbsoluteTargetAddr depending on whether or not just executed instruction is a taken control transfer.

The list of instructions specified for credit and pre-typed table initializers which you can copy and add additional fields (if you use a table driven decoder) will appear in Project2/Arch fetchPC should be initialized so fetchPC=the start virtual address (either 0x0 or taken from the ELF header). The other registers are initialized as described above. The first time the loop is run, nothing should be done by the execution stage. (A tricky way to make this happen is to initialize IR with an SLL $0, $0, 0 instruction which has encoding 0x00000000 and does nothing. It is called a nop instruction.)

Part 2

Study the README file and other material in ~csi402/pub/ElfResources/Demonstration Documentation is linked from the class home page.

Part 3: ‘mapper’

1. mapper runs an interactive command loop with the prompt:
   mapper>
   (one space after the >)

2. Exit when the command is mapper> exit

3. When the command is
   mapper> create segment
   print the prompts below (each followed by a newline); read the indicated information.
   Base Virtual Address?
   (You type in the virtual address (hex) of the first byte of a new segment.)
   Segment Size?
   (You type in the size (hex) in bytes of the new segment.)
   Permissions(rwx)?
   (You type three characters xyz where x is r or -, y is w or - and z is x or -.)
   After receiving the data, use new unsigned long[(you figure out)] to allocate the minimum number of aligned 4-byte words to implement the segment. Install the information needed to translate virtual addresses to C++ char or unsigned long pointers so that bytes or aligned words of the segment can be accessed. The data in the translation table is shown below.

4. When the command is
   mapper> show table
   Print information about each segment following this example:
[0] vbase=003ff000 vlimit=00400280 prot=r-x (basep=023a54e0 size=00001280)
[1] vbase=10010280 vlimit=100102a4 prot=rw- (basep=023a7efc size=00000024)
[2] vbase=7fffffff vlimit=80000000 prot=rwx (basep=023a7fff4 size=00000400)

Formats will appear in Messages/Mmemory.h
Here, vbase is the address of the first byte in the segment, vlimit is the address of
the first (unallocated) byte after the segment and prot gives the segment’s protection
attributes (Read(r), Write(w) and Execute(x)). The data in parentheses are information
about how your program implements the segment: The values are what is returned by
the new operator and the number of bytes allocated.

Your virtual memory simulation code must use the basep and vbase values to map virtual
addresses to C++ pointers, and the vlimit value to test if a given virtual address is in a
given segment.

5. When the command is
mapper> translate 0xvvvvvvv
(here a virtual address in hexadecimal is given) print the corresponding C++ pointer
value as shown:
paddr=0xpppppppp
if a segment containing 0xvvvvvvvv exists, otherwise print
segmentation error

6. When you (instead of the cpu program) type one of the five memory requests in specification (7) of part 1, make the mapper print the response specified there if the access is
legal. Otherwise, print the first of the messages below that applies:
segmentation error if the virtual address is not in any of the segments.
access error if the kind of access: read(r), write(w) or instruction fetch, called execute(x), is not permitted by the protections of the segment containing the given virtual
address.
alignment error if a word (data or instruction) access is attempted at a virtual address that is not word aligned (not a multiple of 4).

You and the graders should test this by seeing if the data you write into virtual memory
is the same as the data you later read from the same address.

7. When segment creation is requested, check if the new segment overlaps any previously
created segment. If there is an overlap, print
Overlap with segment n
where n is the segment number of the first overlapped segment. Do not create the new
segment in this case.

8. If your virtual memory mapper does not support alignment to pages, print
0x4
when the command
mapper> show pagesize
is given.
9. Support the requirement of MIPS and other computers that segment starting addresses and segment lengths must be a multiple of the hardware page size. Assume that the page size is 0x40 (64 bytes, unrealistically small), so print 0x40 after the show pagesize command. Then, regardless of the alignment of a requested base virtual address, make the base virtual address you use be the largest multiple of the page size so the segment contains the requested virtual address. Regardless of the alignment of the address after the requested segment end, make the segment end at a page boundary in the similar way. (This means a segment will be larger than the requested size if the requested virtual base address and/or (base address + requested size) are not aligned to the page size.)

Part 4: ‘‘elftool’’

Read gABI pages 4-1 to 4-9 (concentrating on the “execution view”), gABI pages 5-1 to 5-7 and mABI pages 5-1, 5-2 and 5-5. Also, read the file on the CSC cluster /usr/include/sys/elf.h as needed.

1. Open for reading the file named by the last command line argument. If no filename is given, print the message No filename given., newline, and exit. If the opening fails, run the code

```c
#include <stdio.h> //for the declaration of perror
...
perror("Can not open given file");
exit(1);
```

2. Test your program in a case when the named file doesn’t exist and the case when it does exist but is unreadable. Make it unreadable with the command chmod 000 filename. Read documentation for perror(). Observe the difference in what perror() prints for your two tests; figure out why. Try to determine the relevant errno values.

3. The given file should be an ELF file. Read the ELF header and print all of its fields. Specification (gABI 4-4 to 4-9, pdf.46-51) explains this data. Print it in the order that the data appear in the file using the macros given in MELFHeader.h of the project Messages directory.

4. Print each “program header” from the program header table. (The program headers specify the segments to be loaded. Formats to be supplied in Messages/MELFSegs.h

It is important of course to use the size of the program header table that is given in the ELF header to allocate the memory in which to read the program header table. Watch out for what quantity the “size” represents; you will also need to use the e_phentsize header entry too. (also the file offset, of course)

5. Allocate memory for each loadable segment. Read each loadable segment into memory.
6. Print the contents of each loadable segment using the \texttt{dump()} subroutine. Read \texttt{dump.h}, \texttt{demo-dump.C} and use \texttt{dump.h} and \texttt{dump.C} from \texttt{Project2/Messages} Verify that your program prints the same data as \texttt{objdump} [details to be demonstrated in lecture].

7. Make sure that the “Virtual Address” report from \texttt{dump()} gives the correct virtual address for the start of the segment.

8. Make sure that if a segment’s file size is smaller than its memory size, the amount of memory allocated is the memory size and the bytes between the file data and the end of the segment are initialized to 0.

\textbf{Part 5: “simmips”}

1. Recognize and record the \texttt{--state-trace} and \texttt{--decode-trace} options on the command line; their effects are the same as in Part 1. Also, recognize and record option \texttt{--verify-load} Ignore unrecognized or repeated options. (You can add your own options to activate debugging code if you like.) The last command line argument should be the file name of an ELF MIPS executable file. Make the program try to open this file as in Part 4. Print the error messages specified there for errors and exit.

2. (Setup and load segments) \texttt{simmips} will allocate memory and setup tables in order to implement and load the loadable segments (\texttt{gABI 5-3}) specified in the ELF file, and to implement the stack segment as described in (\texttt{mABI 3-23 to 24 }). Make your simulator stack segment 1KByte ($2^{10}$) in size with its highest address (bottom) 0xfffffffe. The stack segment should initially contain all 0s.

Since each stack frame should be aligned to 8 bytes aligned, make the initial value of \texttt{\$sp} (i.e., \$29 be \texttt{7ffffff}). (The bottom 3 bits of this address are 0.)

The stack grows toward lower addresses. See (\texttt{mABI 3-30 to 3-35}) for details about the initial stack contents specified for processes to run in a MIPS system. (You do not have to implement all of this—Just provide a usable stack segment so compiled C programs can be simulated. One purely optional ambition is to copy the the argument list and environment from the \texttt{simmips} process into the simulated MIPS stack as specified in \texttt{mABI} so you can simulate the running of MIPS processes that operate on their command line arguments and environment. Another idea for independent exploration is to allow the stack segment to increase in size dynamically.)

3. If the \texttt{--verify-load} option was given, print the table specified under the \texttt{show table} command of Part 3. Then, for each segment, run the \texttt{dump()} subroutine provided in the \texttt{Project2 Messages} directory.

4. Initialize the (simulated) registers as in Part 1, except initialize \texttt{\$sp} which is \$29, the stack pointer to the address of the last item on the stack.

5. Run the Simulation Loop.
Resources

Some of the detailed specifications you will need for this project are to be found in the MipsProc, gABI and mABI specification documents listed below. The specification references that are part of the project assignment are listed with reference NUMBERS in the Specification References section of the assignment.

- This assignment sheet.
- Beck Chapters 2 and 3 for general information about assemblers and linkers/loaders. Read as referred to in the lecture and as needed. See pages 29-32 for an introduction to another RISC architecture called SPARC—It is the architecture of the ECL and SUNY Albany Unix Computing Cluster computers. See pages 105-108 for concepts of ELF file/assembler sections (as opposed to segments), loading and delayed branches. Delayed branches seem weird but are necessary for you to implement so you can test your simulators on real programs.
- Hennessy and Patterson’s Computer Organization and Design text used in CSI333, for MIPS machine/assembly language, pipelined processors, and spim.
- Web links and other references on the Textbooks and other Reading Resources “Resources” page linked from the CSI402 home page.
- The MIPS R4400 Microprocessor User’s Manual, 2nd edition by J. Heinrich, 1994. This is available in .pdf and postscript format on the Web linked from Resources. Two paper copies are on course CSI402 library reserve. References to this document will be denoted by MipsProc.
- System V Application Binary Interface Specification, generic version 3.1, chapters 3, 4 and 5. References to this document will be denoted by gABI. There is a link from the course Resources web page.
- The MIPS Assembly Language Reference Manual linked from the course Resources web page. References to it are denoted by MipsAsm.
- System V Application Binary Interface Specification, MIPS RISC Processor Supplement, version 3.0, chapters 3, 4, and 5. References to this document will be denoted by mABI. There is a link from the Resources web page for this too.
- Files in “csi402/pub/Project2” (class account), also accessible via the Web.

To make the specification more definite and facilitate automated grading, messages and data must be printed by using the macros defined in header files named Msomething.h as the “argument” to the “put-to” or stream insertion operator <<. See the “demo-” programs for examples. These files are found in the Messages subdirectory. See also demo-dump.C for tips on doing the C++ memory allocation for the simulator memory. (Information: Externally defined format strings, including error messages and online documentation, provide a way to easily produce versions of software for other (natural)
languages from the version originally developed for one language. A more modern way to achieve this is with what are sometimes called Resource Databases: Examples are the X Window System Resource Database, Windows Resource files, WindowsNT/95 Registry, similar facilities on the MacIntosh, and facilities in various graphical user interface libraries. With generic resource database, application programs refer to things like format strings and documentation by ”keys” and the database stores the actual string, etc. values. Libraries are provided for the application to retrieve values by presenting the key. Resources will also be used automatically to configure properties of windows.)

* Local SUNYA course conversation newsgroup sunya.class.csi402 (official announcements are posted on sunya.class.csi402.announce)
### Specification References

<table>
<thead>
<tr>
<th>Number</th>
<th>Description and additions</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opt</td>
<td>intro. to RISC.</td>
<td>MipsProc p1-4; pdf30-33.</td>
</tr>
<tr>
<td>M3</td>
<td>instruction groups</td>
<td>MipsProc p:15,A-2; pdf:45,470</td>
</tr>
<tr>
<td>M5</td>
<td>data formats. Do bigendian or MSB ordering only.</td>
<td>MipsProc p:24, pdf:54</td>
</tr>
<tr>
<td>INFO</td>
<td>little endian or LSB ordering</td>
<td>MipsProc p:24-25; pdf:54-55</td>
</tr>
<tr>
<td>M6</td>
<td>alignment constraints</td>
<td>MipsProc p:25; pdf:55. Top only.</td>
</tr>
<tr>
<td>INFO</td>
<td>operating modes</td>
<td>MipsProc p:32; pdf:62</td>
</tr>
<tr>
<td>M11</td>
<td>load/store addressing mode</td>
<td>MipsProc p:37; pdf:67</td>
</tr>
<tr>
<td>M12</td>
<td>access type determined from address</td>
<td>MipsProc p:37; pdf:67</td>
</tr>
<tr>
<td>M13</td>
<td>permissible combinations. Do word and byte only.</td>
<td>MipsProc p:38; pdf:68</td>
</tr>
<tr>
<td>M15</td>
<td>delayed jump and branch</td>
<td>MipsProc p:41; pdf:71</td>
</tr>
<tr>
<td>M16</td>
<td>jump;jump and link target address</td>
<td>MipsProc p:41; pdf:71</td>
</tr>
<tr>
<td>M17</td>
<td>jump register; jump and link register target address</td>
<td>MipsProc p:41; pdf:71</td>
</tr>
<tr>
<td>M18</td>
<td>branch (conditional control transfer) target address</td>
<td>MipsProc p:41; pdf:71</td>
</tr>
<tr>
<td>Opt</td>
<td>conditional branch likely instructions are not required</td>
<td>MipsProc p:41; pdf:71</td>
</tr>
<tr>
<td>Opt</td>
<td>MIPS CPU pipeling</td>
<td>MipsProc Chapter 3; Patterson and Hennessy Computer Organization and Design</td>
</tr>
<tr>
<td>INFO</td>
<td>TLB managed by software</td>
<td>MipsProc Chapter 4</td>
</tr>
<tr>
<td>Number</td>
<td>Description and additions</td>
<td>Reference</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>M24</td>
<td>detailed instruction specifications</td>
<td>refer to MipsProc p:A-11 to A-180; pdf:479-648 as needed. Since the instruction set is so regular, the detail pages for most of the instructions you will implement are not needed.</td>
</tr>
<tr>
<td>M25</td>
<td>the opcode table. Use CAPITALIZED instruction names for hardware instructions. Assembler names and convenience instructions such as nop and move are in lower case.</td>
<td>MipsProc p:A-181; pdf:649</td>
</tr>
<tr>
<td>M26</td>
<td>Unimplemented opcodes cause an unimplemented instruction error message</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>ELF file layout</td>
<td>gABI 4-1 to 4-2; mABI 5-1 (fig. 5-1). Read about paging for info only.</td>
</tr>
<tr>
<td>E2</td>
<td>data types</td>
<td>gABI 4-3</td>
</tr>
<tr>
<td>E3</td>
<td>ELF header</td>
<td>gABI 4-4 to 4-6</td>
</tr>
<tr>
<td>E4</td>
<td>ELF ident</td>
<td>gABI 4-7 to 4-8</td>
</tr>
<tr>
<td>E5</td>
<td>byte order or endianess</td>
<td>gABI 4-9</td>
</tr>
<tr>
<td>E6</td>
<td>MIPS e_ident</td>
<td>mABI 4-1 (figure 4-1 only)</td>
</tr>
<tr>
<td>E7</td>
<td>no overlapping virtual addresses</td>
<td>mABI 4-9</td>
</tr>
<tr>
<td>E8</td>
<td>gp relative addressing</td>
<td>mABI 4-11</td>
</tr>
<tr>
<td>M30</td>
<td>32 bit wide virtual address space, user segments are in 31 bit wide virtual address space only. cpu will recognize 2 addresses in unmapped, uncached kernel mode accessible space to implement the SOD</td>
<td>MipsProc p:75-76; pdf:105-106</td>
</tr>
</tbody>
</table>