6.823 Computer System Architecture
EDSACjr

http://csg.csail.mit.edu/6.823/

The first computer architects did not know exactly what was needed in their machines. They did know, however, that parts were expensive and unreliable. Thus, these pioneers developed architectures that minimized hardware while attempting to provide sufficient functionality to programmers. One of the first electronic computers, EDSAC, had a single accumulator and only absolute addressing of memory. Since one could reference memory only by an address listed explicitly within a program, self-modifying code was essential.

We now know that register based addressing and indirection make assembly level programming and compilation a lot easier. This lesson was learned, however, after programmers spent nearly five years programming absolute addressing machines. Here we introduce an instruction set that gives the functionality of EDSAC without some of the more obscure opcodes. This instruction set, named the EDSACjr, is described in Table H1-1. In the notation used in the table below, M[*x*] stands for the contents of the memory location addressed by *x*. Accum refers to the accumulator. ← signifies that data is transferred (copied) from the location to the right of the ← to the location on the left. The immediate variable *n* is an address or a literal depending on the context. The EDSACjr architecture allows programmers to put constants at any point in the memory when a program is loaded.

Opcode	**Description**	**Bit Representation**
ADD *n*	Accum ← Accum + M[*n*]	00001 *n*
SUB *n*	Accum ← Accum - M[*n*]	10000 *n*
STORE *n*	M[n] ← Accum	00010 *n*
CLEAR	Accum ← 0	00011 00000000000
OR *n*	Accum ← Accum | M[*n*]	00000 *n*
AND *n*	Accum ← Accum & M[*n*]	00100 *n*
SHIFTR *n*	Accum ← Accum shiftr *n*	00101 *n*
SHIFTL *n*	Accum ← Accum shiftl *n*	00110 *n*
BGE *n*	If Accum ≥ 0 then PC ← *n*	00111 *n*
BLT *n*	If Accum < 0 then PC ← *n*	01000 *n*
END	Halt machine	01010 00000000000

Table H1-1: The EDSACjr instruction set

The shifts are arithmetic shifts. All words are 16 bits long. As in EDSAC, instructions are encoded as integers. The first 5 bits are the opcode and the last 11 bits form the immediate field (an 11-bit immediate address addresses up to 2048 words (16-bit) of memory -- twice that of the real EDSAC). Integers are represented in 16 bits, the most significant bit being a sign bit.

Using Macros for EDSACjr

What are Macros?

Macros are assembler directives that allow the programmer to define short names for a sequence of instructions. Once defined, a macro can be used within the code in place of the sequence of instructions, thus saving the programmer time and effort, as well as making the program easier to read. At assembly time, macros are *expanded* according to their definition. That is, each occurrence of a macro is replaced by its corresponding instruction sequence. As described below, macros can have arguments, as well as both global and local labels.

A Simple Example

If you have already tried writing code with the EDSACjr instruction set, you have probably noticed that there is no LOAD instruction for putting the value at a memory location into the accumulator. To do a LOAD, you need to first CLEAR the accumulator, and then ADD the contents of the desired memory location to it. If you need to do a lot of LOADs, it can be quite cumbersome to always have to type this CLEAR-ADD sequence. To make it easier, we can define the following macro for LOAD:

```
.macro LOAD(n)
     CLEAR
     ADD n
.end macro
```

Then, whenever the line LOAD(n) occurs in the code, it will be replaced by the two instructions, CLEAR and ADD as defined. For example:

```
LOAD A          → expands to →     CLEAR
SUB B                              ADD A
                                   SUB B
```

Note how the argument of the macro is used during expansion. It is also possible to use multiple arguments separated by commas.

It is very important to note that a macro is *not* the same as a subroutine or function. A function call involves jumping to and returning from a *single* piece of subroutine code located somewhere in memory, while a macro "call" involves in-place substitution of the macro code. That is, if there are multiple occurrences of a macro, then the macro code is duplicated multiple times, once at each occurrence in the code. This duplication does not happen with function calls. Because of this difference, using macros usually results in more actual code than using a function call. However, it also usually results in slightly faster code, since macros do not have the calling-convention overhead needed by function calls.[1]

Global Labels

Since macros work by simple expansion, you can refer to labels outside the macro, and these labels will be used verbatim as long as the name of the label does not conflict with any of the macro's arguments or with labels defined within the macro. For example, suppose we define the following macro:

```
.macro STOPGE
     BGE stop
.end macro
```

We may then use this macro as follows:

```
        STOPGE       → expands to →          BGE stop
        SUB B                                SUB B
 stop:  ADD A                         stop:  ADD A
```

As shown, global labels are useful for accessing commonly used locations in memory.

Local Labels

You can also define and use labels *within* a macro. During expansion, such local labels will be replaced by a unique label for each instance of the macro. For example, consider the following interesting (silly?) macro:

```
.macro HANGGE
here:
     BGE here  ; if accum >= 0, then loop forever
.end macro
```

When this macro is used, the local labels are expanded as follows:

```
HANGGE        → expands to →     here_1:  BGE here_1
SUB B                                     SUB B
HANGGE                           here_2:  BGE here_2
```

[1] Another interesting difference between macros and function calls is that you cannot write recursive macros. Think of what will happen if you do.

Note that the local label `here` is converted to a unique label during expansion so that multiple instances of a macro do not interfere with each other.

It is also possible to place a label at the end of a macro definition, even if there is no instruction at that position. Such a label would point to the instruction immediately following the macro call in the main code. For example, we can define a conditional ADD macro as follows:

```
.macro ADDGE n       ; this macro only adds if accum >= 0
     BLT donothing  ; if accum < 0,
                    ; then just go to instruction after
                    ; macro
     ADD n          ; else accum <- accum + M[n]
donothing:
.end macro
```

It can be used as follows:

```
ADDGE A     → expands to →                    BLT donothing_3
SUB B                                         ADD A
                              donothing_3:   SUB B
```

6.823 Computer System Architecture
CISC ISA – x86jr

http://csg.csail.mit.edu/6.823/

x86 has a CISC-style instruction set with variable-length instructions. In the x86 architecture, each instruction is capable of performing one or more simpler instructions called micro-operations (μops). It also supports several complex addressing modes.

We introduce a (very small) subset of the x86 instruction set in the following table. (Interested readers are referred to the Intel's website for full details.)

Instruction	**Operation**	**OF**	**SF**	**Length**
`add` R_{DEST}, R_{SRC}	$R_{DEST} \leftarrow R_{SRC} + R_{DEST}$	M	M	2 bytes
`cmp imm32,` R_{SRC2}	`Temp` $\leftarrow R_{SRC2}$ `- MEM[imm32]`	M	M	6 bytes
`inc` R_{DEST}	$R_{DEST} \leftarrow R_{DEST} + 1$	M	M	1 byte
`jmp label`	jump to the address specified by `label`			2 bytes
`jl label`	if `(SF≠OF)` jump to the address specified by `label`	T	T	2 bytes
`xor` R_{DEST}, R_{SRC}	$R_{DEST} \leftarrow R_{DEST}$ `xor` R_{SRC}	O	M	2 bytes

Table H2-1: Simple x86 instruction set (x86jr)

Notice that the jump instruction `jl` (jump if less than) depends on SF and OF, which are status flags. Each instruction affects them in different ways based on the result of its computation: "M" indicates the instruction modifying (writing) the status flag, "T" the instruction testing (reading but not writing) it, and "O" the instruction resetting it. A blank (as in `jmp` instruction) means that the instruction does not affect the status flag. Some instructions, like the `cmp` instruction, perform a computation and set status flags, but do not return any result.

The meanings of the status flags are given in the following table:

Name	**Purpose**	**Condition Reported**
OF	Overflow	Result exceeds positive or negative limit of number range
SF	Sign	Result is negative (less than zero)

Table H2-2: Status flags

The following table shows how different values in the operands of `cmp` influence those status flags in a 32-bit machine:

R_{SRC2}	`MEM[imm32]`	**OF**	**SF**
0x00000001	0x00000000	0	0
0x00000000	0x00000001	0	1
0x7FFFFFF (positive limit)	0xFFFFFFF (-1)	1	0
0x80000000 (negative limit)	0x00000001	1	1

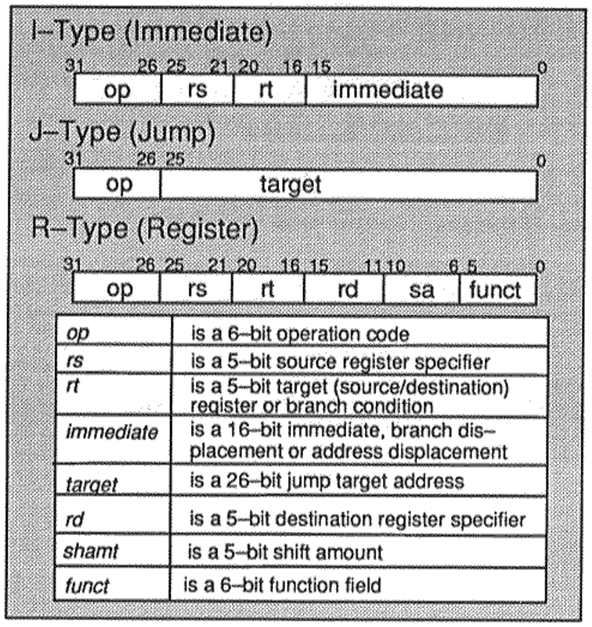
Table H2-3: Operands vs Status flags

6.823 Computer System Architecture
RISC ISA – MIPS32

http://csg.csail.mit.edu/6.823/

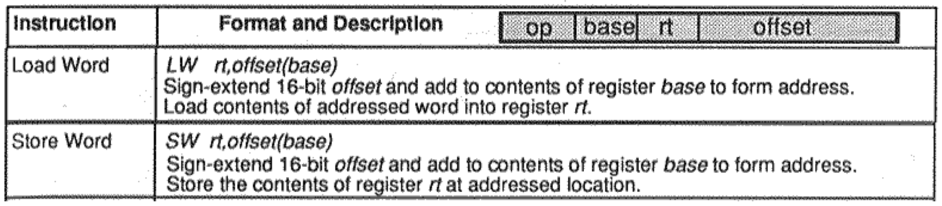
Here is a brief summary of the MIPS instructions used in this course. All general-purpose registers (GPRs) are assumed to be 32 bits. (R0 is hardwired to zero.) For more information, please check out either Section 2.12 or Appendix C (online) of the Hennessy and Patterson book. Unlike the MIPS64 architecture in the Hennessy & Patterson book, we use 32-bit GPRs instead of 64-bit GPRs throughout this course.

Instruction Formats



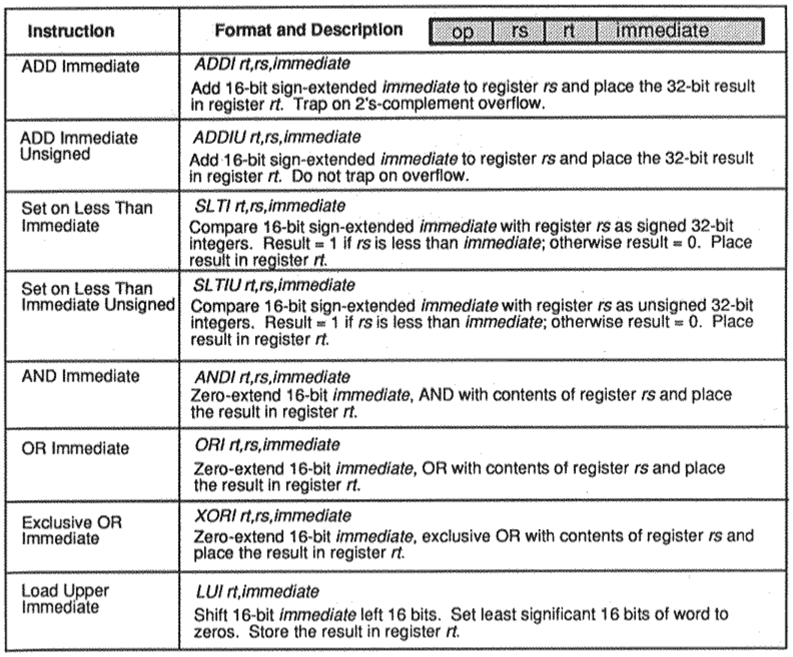
op	is a 6–bit operation code
rs	is a 5–bit source register specifier
rt	is a 5–bit target (source/destination) register or branch condition
immediate	is a 16–bit immediate, branch displacement or address displacement
target	is a 26–bit jump target address
rd	is a 5–bit destination register specifier
shamt	is a 5–bit shift amount
funct	is a 6–bit function field

Load and Store Instructions

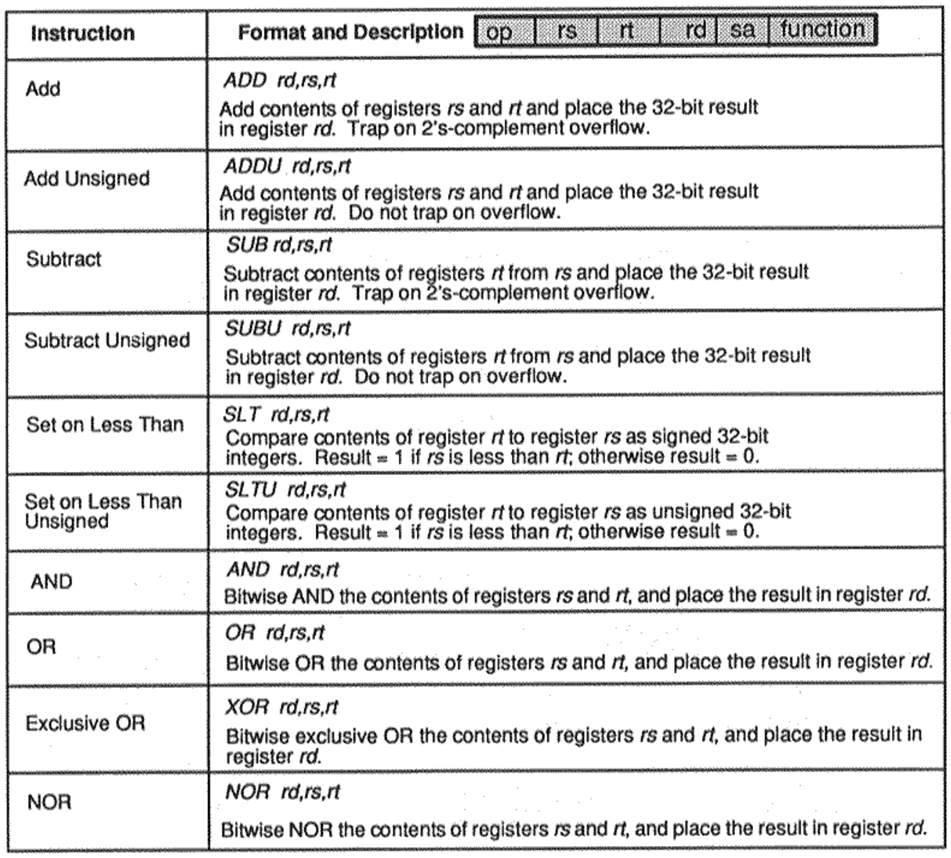


Instruction	Format and Description (op | base | rt | offset)
Load Word	*LW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Load contents of addressed word into register *rt*.
Store Word	*SW rt,offset(base)* Sign-extend 16-bit *offset* and add to contents of register *base* to form address. Store the contents of register *rt* at addressed location.

ALU Instructions



Instruction	Format and Description | op | rs | rt | immediate |
ADD Immediate	*ADDI rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result in register *rt*. Trap on 2's-complement overflow.
ADD Immediate Unsigned	*ADDIU rt,rs,immediate* Add 16-bit sign-extended *immediate* to register *rs* and place the 32-bit result in register *rt*. Do not trap on overflow.
Set on Less Than Immediate	*SLTI rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
Set on Less Than Immediate Unsigned	*SLTIU rt,rs,immediate* Compare 16-bit sign-extended *immediate* with register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *immediate*; otherwise result = 0. Place result in register *rt*.
AND Immediate	*ANDI rt,rs,immediate* Zero-extend 16-bit *immediate*, AND with contents of register *rs* and place the result in register *rt*.
OR Immediate	*ORI rt,rs,immediate* Zero-extend 16-bit *immediate*, OR with contents of register *rs* and place the result in register *rt*.
Exclusive OR Immediate	*XORI rt,rs,immediate* Zero-extend 16-bit *immediate*, exclusive OR with contents of register *rs* and place the result in register *rt*.
Load Upper Immediate	*LUI rt,immediate* Shift 16-bit *immediate* left 16 bits. Set least significant 16 bits of word to zeros. Store the result in register *rt*.

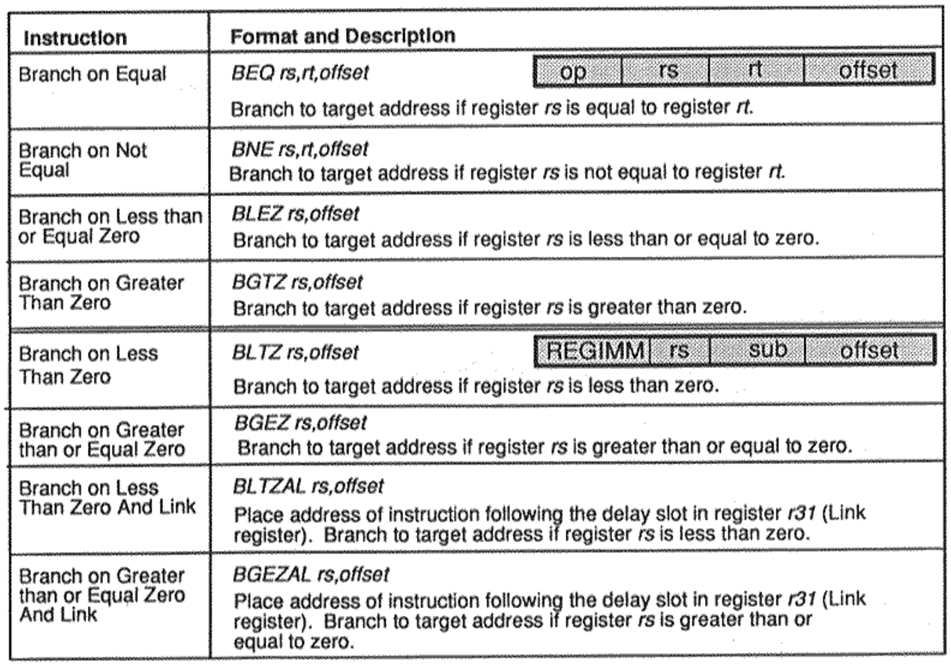


Instruction	Format and Description | op | rs | rt | rd | sa | function |
Add	*ADD rd,rs,rt* Add contents of registers *rs* and *rt* and place the 32-bit result in register *rd*. Trap on 2's-complement overflow.
Add Unsigned	*ADDU rd,rs,rt* Add contents of registers *rs* and *rt* and place the 32-bit result in register *rd*. Do not trap on overflow.
Subtract	*SUB rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 32-bit result in register *rd*. Trap on 2's-complement overflow.
Subtract Unsigned	*SUBU rd,rs,rt* Subtract contents of registers *rt* from *rs* and place the 32-bit result in register *rd*. Do not trap on overflow.
Set on Less Than	*SLT rd,rs,rt* Compare contents of register *rt* to register *rs* as signed 32-bit integers. Result = 1 if *rs* is less than *rt*; otherwise result = 0.
Set on Less Than Unsigned	*SLTU rd,rs,rt* Compare contents of register *rt* to register *rs* as unsigned 32-bit integers. Result = 1 if *rs* is less than *rt*; otherwise result = 0.
AND	*AND rd,rs,rt* Bitwise AND the contents of registers *rs* and *rt*, and place the result in register *rd*.
OR	*OR rd,rs,rt* Bitwise OR the contents of registers *rs* and *rt*, and place the result in register *rd*.
Exclusive OR	*XOR rd,rs,rt* Bitwise exclusive OR the contents of registers *rs* and *rt*, and place the result in register *rd*.
NOR	*NOR rd,rs,rt* Bitwise NOR the contents of registers *rs* and *rt*, and place the result in register *rd*.

Instruction	Format and Description [op | rs | rt | rd | sa | function]
Shift Left Logical	*SLL rd,rt,sa* Shift the contents of register *rt* left by *sa* bits, inserting zeros into the low order bits. Place the 32-bit result in register *rd.*
Shift Right Logical	*SRL rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, inserting zeros into the high order bits. Place the 32-bit result in register *rd.*
Shift Right Arithmetic	*SRA rd,rt,sa* Shift the contents of register *rt* right by *sa* bits, sign-extending the high order bits. Place the 32-bit result in register *rd.*
Shift Left Logical Variable	*SLLV rd,rt,rs* Shift the contents of register *rt* left. The low order 5 bits of register *rs* specify the number of bits to shift left; insert zeros into the low order bits of *rt* and place the 32-bit result in register *rd.*
Shift Right Logical Variable	*SRLV rd,rt,rs* Shift the contents of register *rt* right. The low order 5 bits of register *rs* specify the number of bits to shift right; insert zeros into the high order bits of *rt* and place the 32-bit result in register *rd.*
Shift Right Arithmetic Variable	*SRAV rd,rt,rs* Shift the contents of register *rt* right. The low order 5 bits of register *rs* specify the number of bits to shift right; sign-extend the high order bits of *rt* and place the 32-bit result in register *rd.*

Jump and Branch Instructions

Instruction	Format and Description [op | target]
Jump	*J target* Shift the 26-bit *target* address left two bits, combine with high order four bits of the PC, and jump to the address with a 1-instruction delay.
Jump And Link	*JAL target* Shift the 26-bit *target* address left two bits, combine with high order four bits of the PC, and jump to the address with a 1-instruction delay. Place the address of the instruction following the delay slot in *r31* (*Link* register).
Instruction	**Format and Description [op | rs | rt | rd | sa | function]**
Jump Register	*JR rs* Jump to the address contained in register *rs*, with a 1-instruction delay.
Jump And Link Register	*JALR* rd, rs Jump to the address contained in register *rs*, with a 1-instruction delay. Place the address of the instruction following the delay slot in register *rd.*



Instruction	Format and Description
Branch on Equal	*BEQ rs,rt,offset* — op | rs | rt | offset Branch to target address if register *rs* is equal to register *rt*.
Branch on Not Equal	*BNE rs,rt,offset* Branch to target address if register *rs* is not equal to register *rt*.
Branch on Less than or Equal Zero	*BLEZ rs,offset* Branch to target address if register *rs* is less than or equal to zero.
Branch on Greater Than Zero	*BGTZ rs,offset* Branch to target address if register *rs* is greater than zero.
Branch on Less Than Zero	*BLTZ rs,offset* — REGIMM | rs | sub | offset Branch to target address if register *rs* is less than zero.
Branch on Greater than or Equal Zero	*BGEZ rs,offset* Branch to target address if register *rs* is greater than or equal to zero.
Branch on Less Than Zero And Link	*BLTZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is less than zero.
Branch on Greater than or Equal Zero And Link	*BGEZAL rs,offset* Place address of instruction following the delay slot in register *r31* (Link register). Branch to target address if register *rs* is greater than or equal to zero.

6.823 Computer System Architecture
Cache Implementations

http://csg.csail.mit.edu/6.823/

Direct-mapped Cache

The following diagram shows how a direct-mapped cache is organized. To read a word from the cache, the input address is set by the processor. Then the index portion of the address is decoded to access the proper row in the tag memory array and in the data memory array. The selected tag is compared to the tag portion of the input address to determine if the access is a hit or not. At the same time, the corresponding cache block is read and the proper line is selected through a MUX.

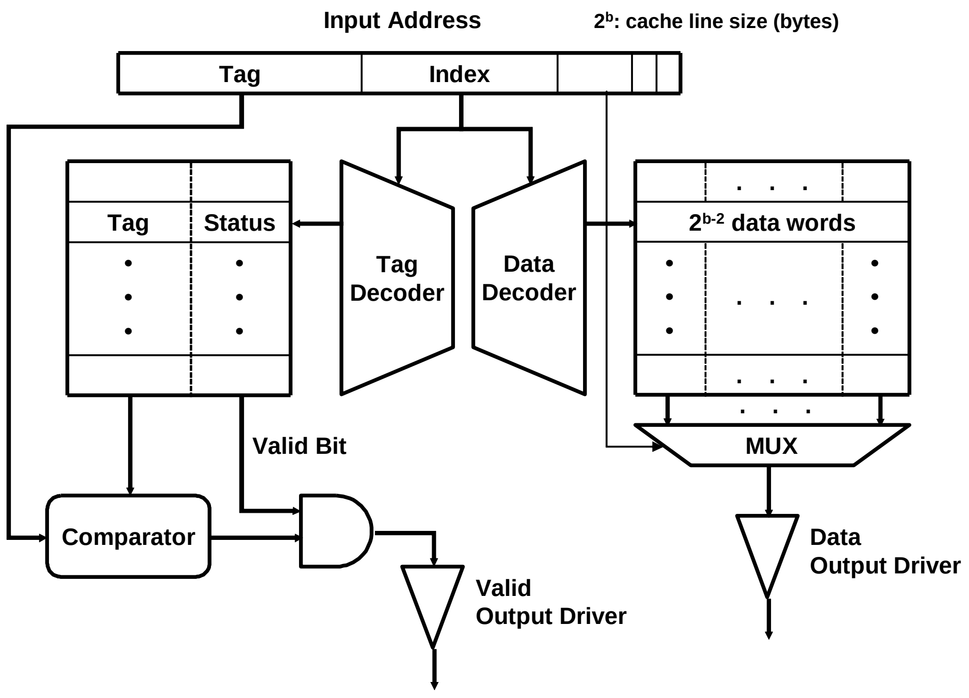


Figure H4-A: A direct-mapped cache implementation

In the tag and data array, each row corresponds to a line in the cache. For example, a row in the tag memory array contains one tag and two status bits (valid and dirty) for the cache line. For direct-mapped caches, a row in the data array holds one cache line.

Four-way Set-associative Cache

The implementation of a 4-way set-associative cache is shown in the following diagram. (An *n*-way set-associative cache can be implemented in a similar manner.) The index part of the input address is used to find the proper row in the data memory array and the tag memory array. In this case, however, each row (set) corresponds to four cache lines (four ways). A row in the data memory holds four cache lines (for 32-bytes cache lines, 128 bytes), and a row in the tag memory array contains four tags and status bits for those tags (2 bits per cache line). The tag memory and the data memory are accessed in parallel, but the output data driver is enabled only if there is a cache hit.

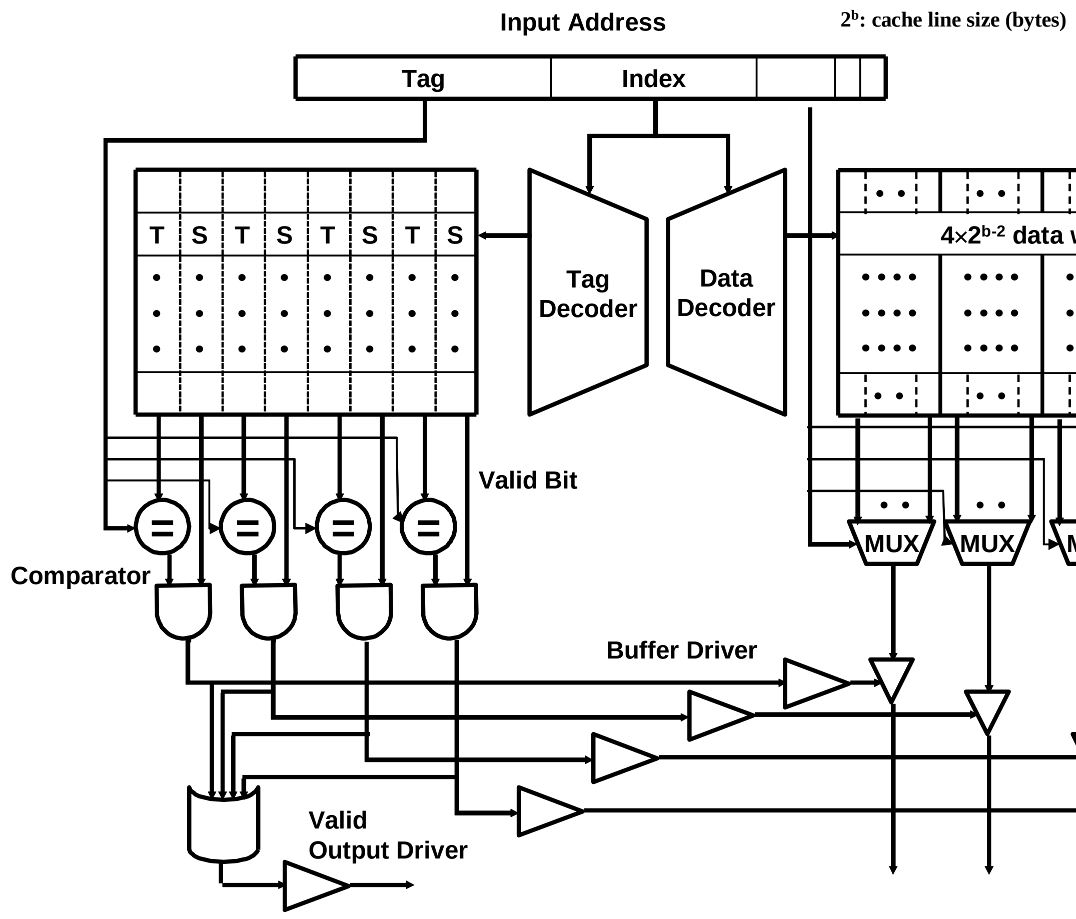


Figure H4-B: A 4-way set-associative cache implementation

6.823 Computer System Architecture
Victim Cache

http://csg.csail.mit.edu/6.823/

Although direct-mapped caches have an advantage of smaller access time than set-associative caches, they have more conflict misses due to their lack of associativity. In order to reduce these conflict misses, N. Jouppi proposed the *victim caching* where a small fully-associative back up cache, called victim cache, is added to a direct-mapped L1 cache to hold recently evicted cache lines.

The following diagram shows how a victim cache can be added to a direct-mapped L1 data cache. Upon a data access, the following chain of events takes place:

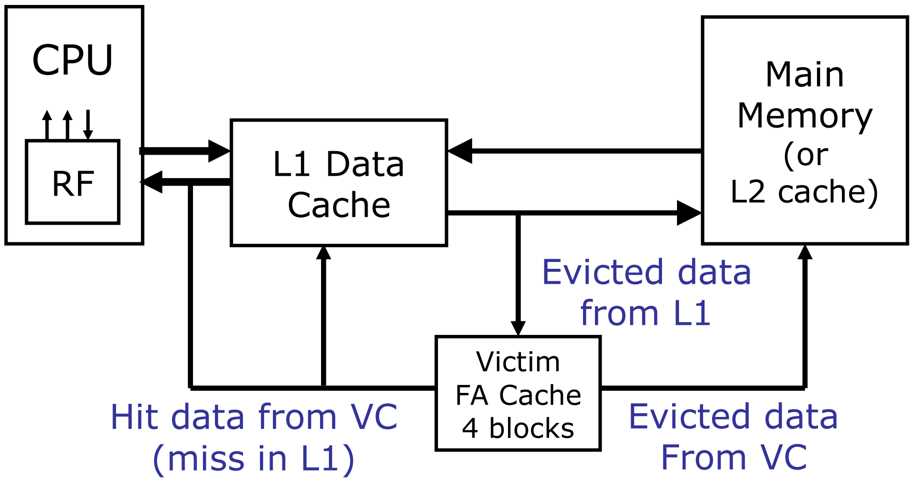


Figure H5-A: A Victim Cache Organization

1. The L1 data cache is checked. If it holds the data requested, the data is returned.
2. If the data is not in the L1 cache, the victim cache is checked. If it holds the data requested, the data is moved into the L1 cache and sent back to the processor. The data evicted from the L1 cache is put in the victim cache, and put at the end of the FIFO replacement queue.
3. If neither of the caches holds the data, it is retrieved from memory, and put in the L1 cache. If the L1 cache needs to evict old data to make space for the new data, the old data is put in the victim cache and placed at the end of the FIFO replacement queue. Any data that needs to be evicted from the victim cache to make space is written back to memory or discarded, if unmodified.

Note that the two caches are *exclusive*. That means that the same data cannot be stored in both L1 and victim caches at the same time.

Reference

1. Norm Jouppi, Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers, in the Proceedings of the 17th *International Symposium on Computer Architecture* (ISCA), pages 364--373, Seattle, Washington, May 1990.

6.823 Computer System Architecture
Virtual Memory Implementation

http://csg.csail.mit.edu/6.823/

Hierarchical Page Table Supporting Variable-Sized Pages

Small fixed-sized pages (e.g. 4 KB) reduce internal fragmentation and the page fault penalty compared to large fixed-sized pages. However, when we run an application with a large working set, they may degrade a processor's performance by incurring a number of TLB misses because of their small *TLB reach*. Therefore, researchers have proposed to support variable-sized pages to increase the TLB reach without losing the benefits of small fixed-sized pages. Many modern processor families (e.g. UltraSparc, PA-RISC, MIPS) and operating systems (e.g. Sun Solaris, SGI IRIX) support this feature.

In this handout, we present an example implementation of variable-sized pages, supporting only two page sizes: 4 KB and 4 MB. Assume that the system uses 44-bit virtual addresses and 40-bit physical addresses. 4KB pages are mapped using a three-level hierarchical page table. 4MB pages are mapped using the first two levels of the same page table. An L2 Page Table Entry (PTE) contains information which indicates if it points to an L3 table or a 4MB data page. All PTEs are 8 Bytes. The following figure summarizes the page table structure and indicates the sizes of the page tables and data pages (not drawn to scale):

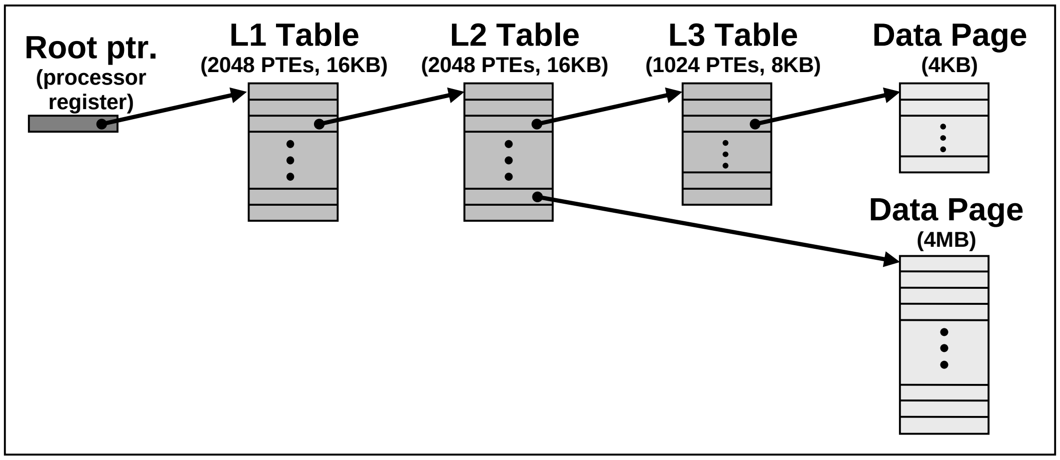


Figure H8-A. Example implementation of variable-sized pages.

Page Table Entries and Translation Lookaside Buffers

Each Page Table Entry (PTE) will map a virtual page number (VPN) to a physical page number (PPN). In addition to the page number translation, each page table entry also contains some permission/status bits.

Bit Name	**Bit Definition**
PPN / DBN	Physical Page Number / Disk Block Number
V (valid)	1 if the page table entry is valid, 0 otherwise
R (resident)	1 if the page is resident in memory, 0 otherwise
W (writable)	1 if the page is writable, 0 otherwise
U (used)	1 if the page has been accessed *recently*, 0 otherwise
M (modified)	1 if the page has been modified, 0 otherwise
S (supervisor)	1 if the page is only accessible in supervisor mode, 0 otherwise

Each entry in the Translation Lookaside Buffer (TLB) has a tag that is matched against the VPN and a TLB Entry Valid bit (note, the TLB Entry Valid bit is not the V bit shown in the table above). The TLB Entry Valid bit will be set if the TLB entry is valid. Each TLB entry also contains all the fields from the page table that are listed above.

A TLB miss (VPN does not match any of the tags for entries that have the TLB Entry Valid bit set) causes an exception. On a TLB miss kernel software will load the page table entry into the TLB and will restart the memory access. (Kernel software can modify anything in the TLB that it likes and always runs in supervisor mode). If the entry being replaced was valid, then the kernel will also write the TLB entry that is being replaced back to the page table.

Hardware will set the used bit whenever a TLB hit to the corresponding entry occurs. Similarly, the modified bit (in the TLB entry) will be set when a store to the page happens.

All exceptions that come from the TLB (hit or miss) are handled by software. For example, the possible exceptions are as follows:

TLB Miss:	VPN does not match any of tags for entries that have the TLB Entry Valid bit set.
Page Table Entry Invalid:	Trying to access a virtual page that has no mapping to a physical address.
Write Fault (Store only):	Trying to modify a read-only page (W is 0).
Protection Violation:	Trying to access a protected (supervisor) page while in user mode.
Page Fault:	Page is not resident.

(Unless noted, exceptions can occur for both loads and stores.)

6.823 Computer System Architecture

Nested Paging

http://csg.csail.mit.edu/6.823/

This handout explores the architectural impact of running a process in a virtual machine, as it relates to virtual memory and address translation.

Platform virtualization (or simply virtualization) is a technique to allow running multiple operating systems over the same physical hardware. Each of these operating systems, called a *guest operating system*, believes it has access to a full physical machine (with a CPU, memory, disks, and other devices). But in fact these resources are emulated—they constitute a *virtual machine*, rather than a physical one, as shown in Figure 1.

The actual physical hardware is in control of the *host operating system*. The host OS in turn can run one or more virtual machines, each with its own emulated resources and guest OS. The host OS abstracts each physical resource and arbitrates access to it among the guest OSs.

Much like an operating system brings the benefits of protection among processes and abstraction of the underlying machine's resources, virtualization achieves protection and abstraction at the level of full virtual machines. Therefore, virtualization has many applications. For example, it can be used to consolidate multiple legacy systems on the same physical hardware; provide stronger levels of security; make it easier to rent physical hardware, as is done in cloud computing; migrate virtual machines among physical machines, etc.

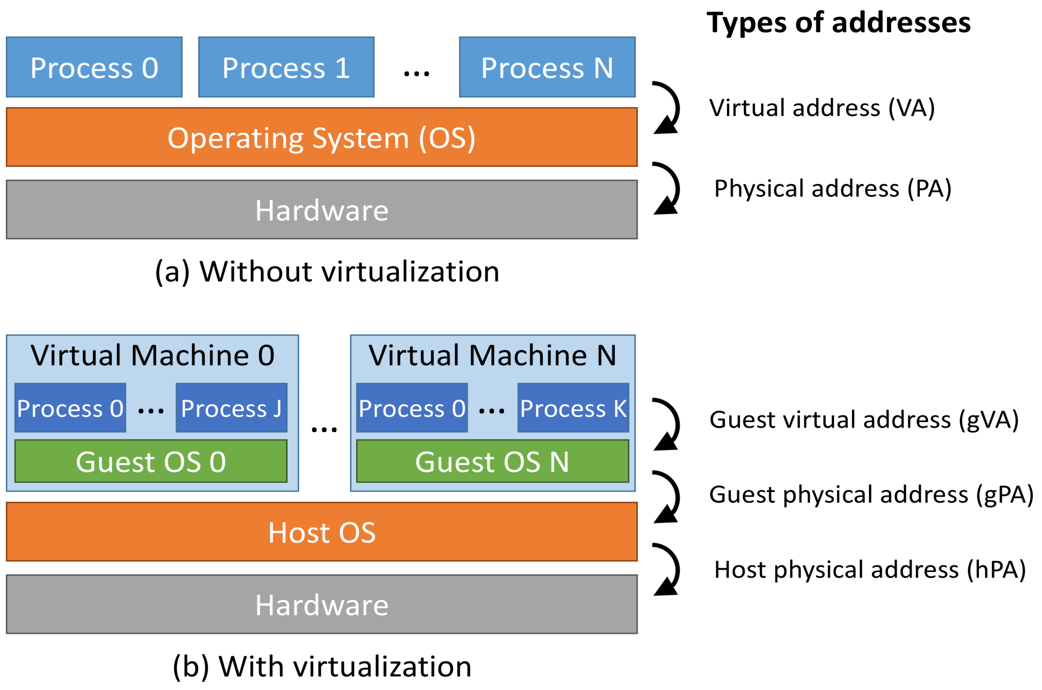


Figure 1: Key elements in systems (a) without and (b) with platform virtualization, and corresponding layers of address translation.

Virtualization requires emulating each of the physical machine's resources. Here we focus on memory. We will discuss other facets of virtualization later in the course.

Each guest OS has access to its *guest memory space*, distinct from the *host memory space.* Processes running on the guest issue loads and stores using virtual addresses, and these are translated to physical addresses, just as we have seen before. However, because the guest memory is distinct from the host memory, this translation alone is insufficient for the host to serve the data to the guest from its real physical memory. In fact, two layers of address translation are required, as shown in Figure 1:

- **`gVA=>gPA`**: guest virtual address to guest physical address translation.
- **`gPA=>hPA`**: guest physical address to host physical address translation.

Every guest physical address appears to the host as a host virtual address.

Nested *paging* is a common design to perform these two address translations. Figure 2 shows an implementation of nested paging using 2-level hierarchical page tables. Below, the host performs `gPA=>hPA` translations through a per-VM 2-level hierarchical page table that **resides in host physical memory**, and whose **PTEs use host physical addresses.**

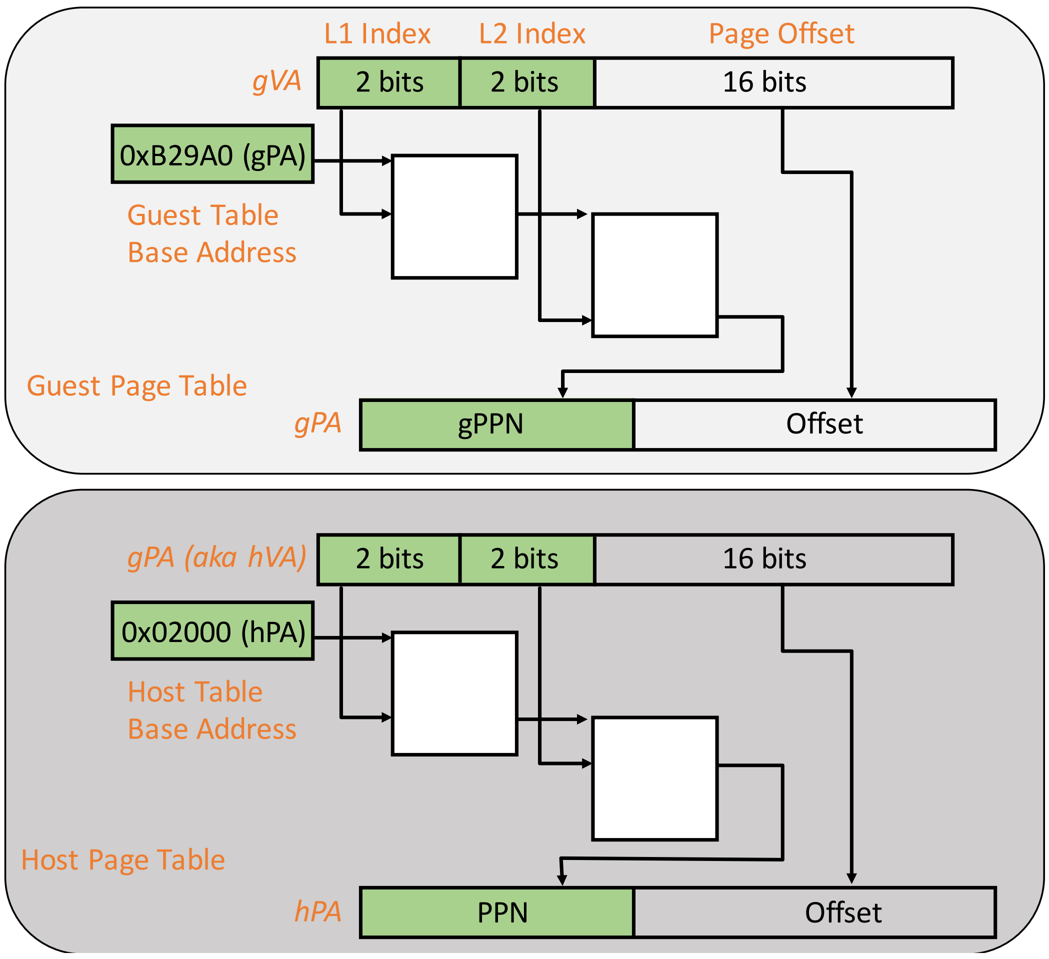


Figure 2: Implementation of *nested paging* with 2-level hierarchical page tables.

The guest also has a 2-level hierarchical guest page table for the `gVA=>gPA` translation, whose **PTEs use guest physical addresses**. We assume the guest L1 and L2 page tables reside in host physical memory. This means that every guest table lookup (which uses a `gPA`) requires an additional `gPA=>hPA` translation. This complexity exists because the guest is unaware it does not run on a real processor!

To make this process efficient, modern systems have hardware support for nested paging. The processor's MMU has two page table base address registers, for the host and guest page tables. The host table base address register holds an `hPA`, and the guest table base address register holds a `gPA`.

The processor's TLB caches only the full `gVA=>hPA` translations. On a TLB miss, the MMU translation performs the nested page table walk. Notice that the lookup of a guest L1 table entry (i.e. `0xB29A0` + `gVA`'s L1index) itself requires a (nested) translation of guest physical address into a host physical address, in order to fetch the relevant guest L1 entry from host physical memory.

6.823 Computer System Architecture
L-MIPS ISA and Single-Cycle Datapath

http://csg.csail.mit.edu/6.823/

Ben Bitdiddle is unhappy with the performance of the MIPS processor discussed in the 6.823 lecture. He wants to extend the MIPS ISA with a new class of instructions, LOAD-ALU (L), described in Table 1. He calls his new ISA the L-MIPS ISA.

31 26	25 21	20 16	15 11	10 0
opcode	rs	rt	rd	imm

Instruction	**Format and Description**
Load and ALU	ALUM rd, offset(base), rt rd ← Mem[(rs) + SignExt(imm)] op (rt) Performs an ALU operation with one register and one memory operand. The effective address of the first operand is computed by adding the contents of register *rs* (base) to the sign-extended 11-bit *immediate* field (offset). The second operand is the contents of register *rt*. eg: ADDM R1, 4(R3), R2 XORM R1, -7(R1), R1

Table 1. L-MIPS LOAD-ALU(L) instruction type.

To implement the L-MIPS ISA, Ben modifies the datapath of the single-cycle MIPS processor as shown in Figure H-1. He adds an address generation unit, and moves the ALU unit after the data memory. Loads, stores and L-type instructions use the address generation unit before querying the memory, while other instructions use the ALU unit after the memory. Note that the address generation unit comprises only an adder.

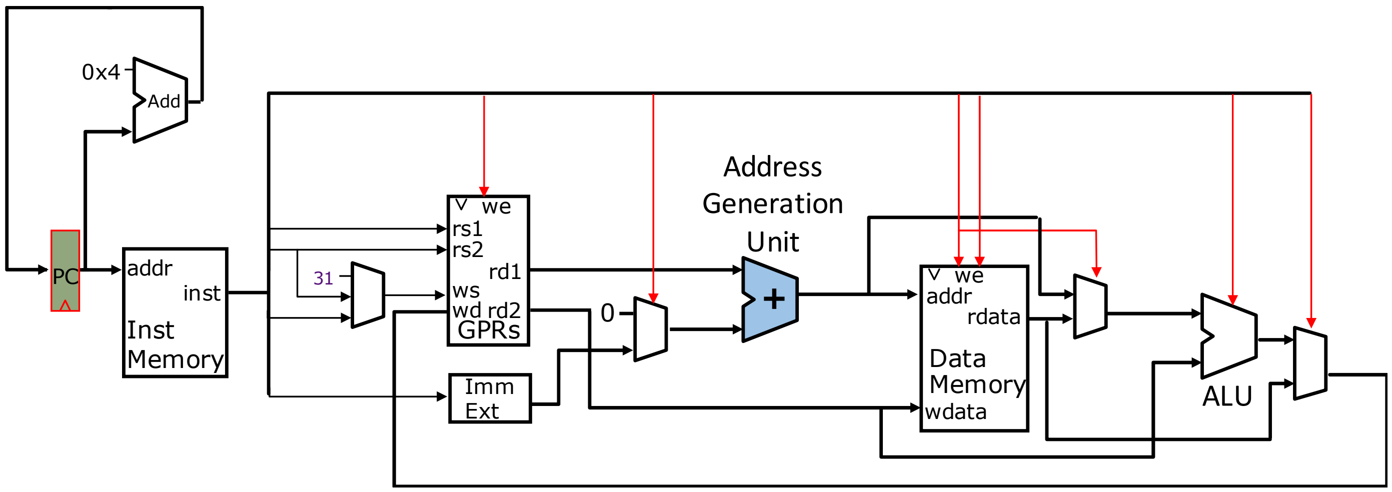


Figure H-1: L-MIPS single-cycle datapath.

6.823 Computer System Architecture

EDSACjr-II

http://csg.csail.mit.edu/6.823/

The EDSACjr (Handout 1) requires using self-modifying code to implement common operations, which adds bookkeeping instructions. EDSACjr-II solves this problem by adding an *index register*, which is used in conjunction with the accumulator to simplify address calculations.

Table 1 summarizes the set of *additional* instructions that EDSACjr-II introduces over EDSACjr. These instructions allow using the index register to construct memory addresses, as well as directly manipulating the index register. The table uses the following notation:

- M[x] stands for the contents of the memory location addressed by x.
- Accum refers to the accumulator, and IX refers to the index register.
- ← signifies that data is transferred (copied) from the location to the right of the ← to the location on the left.
- The immediate variable n is an address or a literal depending on the context.

Note that these instructions are *in addition* to the original set of instructions for EDSACjr – i.e. EDSACjr-II can still perform instructions that do not involve the index register such as ADD *n*, SUB *n*, etc.

Opcode	**Description**
ADDIX *n*	Accum ← Accum + M[*n* + IX]
SUBIX *n*	Accum ← Accum - M[*n* + IX]
LOADIX *n*	Accum ← M[*n* + IX]
STOREIX *n*	M[*n* + IX] ← Accum
ADDi *n*	IX ← IX + *n*
SUBi *n*	IX ← IX - *n*
LOADi *n*	IX ← M[*n*]
STOREi *n*	M[*n*] ← IX
BGEi *n*	If IX ≥ 0 then PC ← *n*
BLTi *n*	If IX < 0 then PC ← *n*
BEQi *n*	If IX == 0 then PC ← *n*
END	Halt machine

Table 1. Additional instructions that EDSACjr-II introduces over EDSACjr.