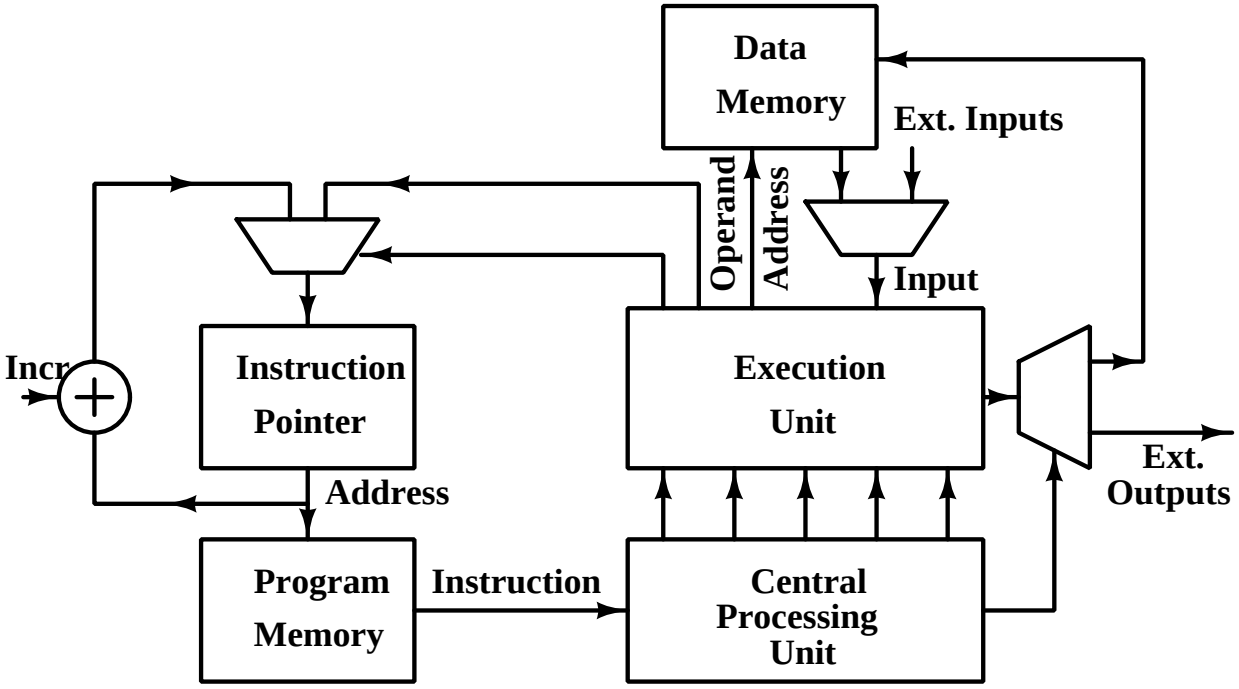
Designing a Processor

Dinesh Sharma

EE Department
IIT Bombay, Mumbai

April 1, 2021

Review



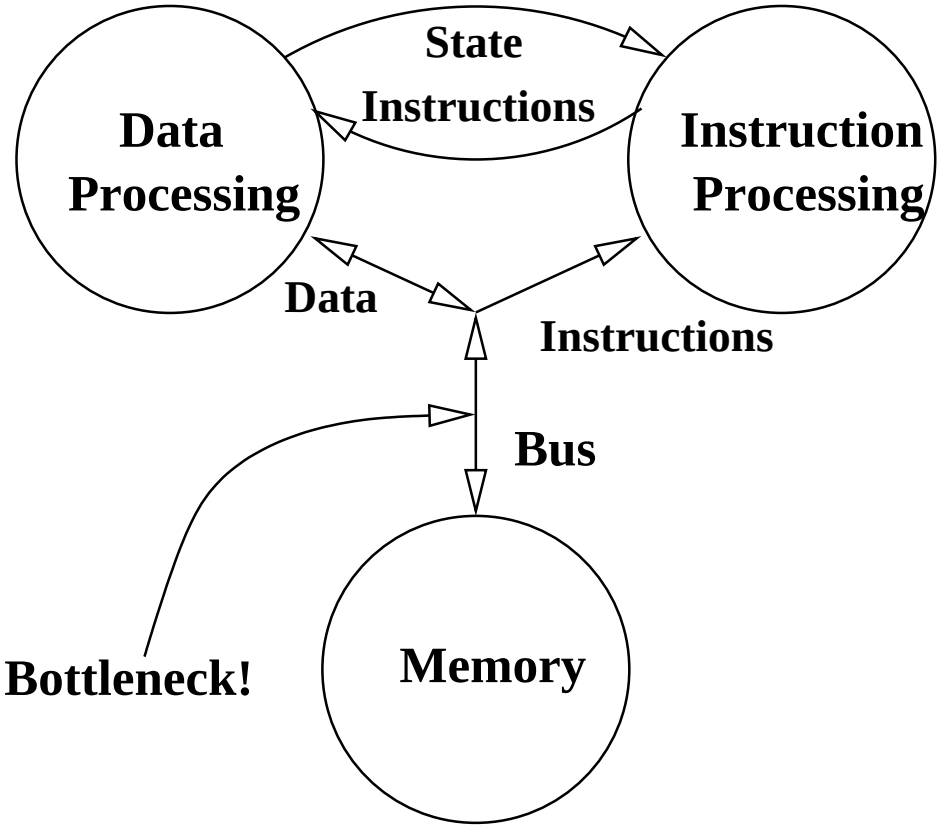
We had seen the evolution of a general purpose digital circuit into a microprocessor in the earlier lectures.

Fetch - Decode - Execute

Implementation of the circuit can be partitioned into a control path and a data path.

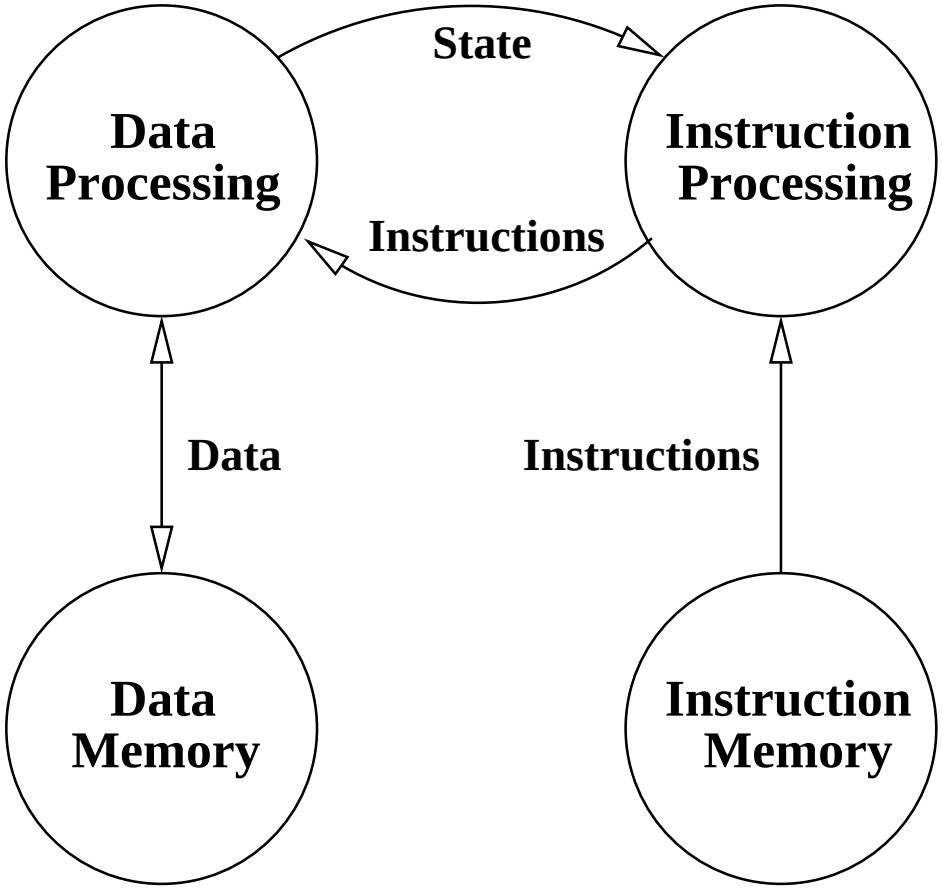
Control path circuits decide the flow of program.
Data path circuits implement the data operations required for the program.

Von Neumann Architecture



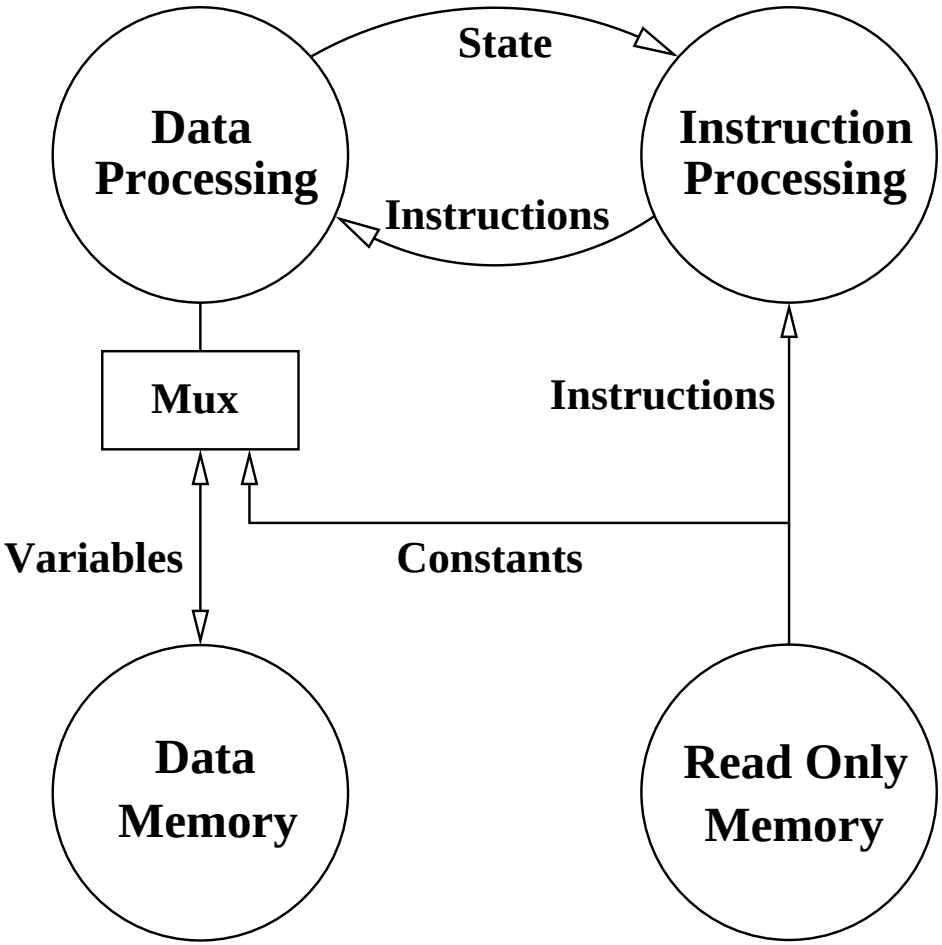
- A common bus is used for data as well as instructions.
- The system can become ‘bus bound’.

Harvard Architecture



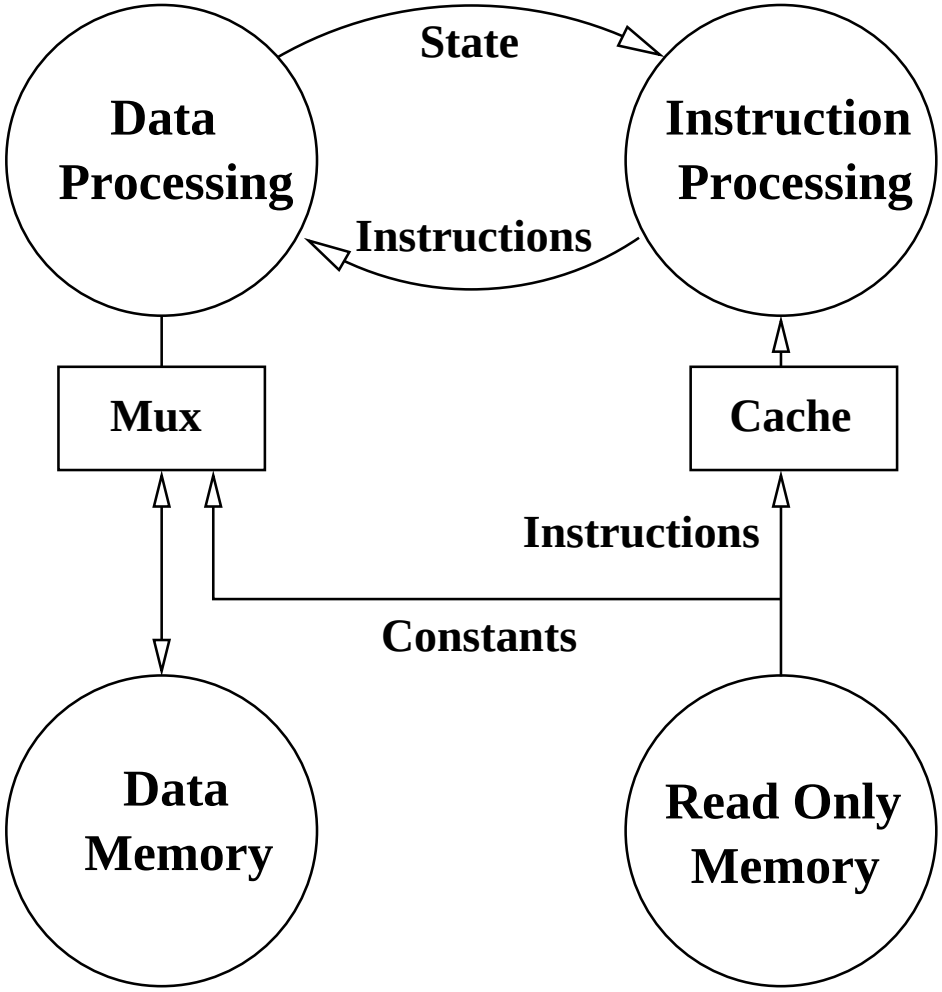
- Separate data and instruction paths
- Good performance
- Needs 2 buses $\rightarrow$ expensive!
- Traffic on the buses is not balanced.
- Instruction bus may remain idle.

Modified Harvard Architecture



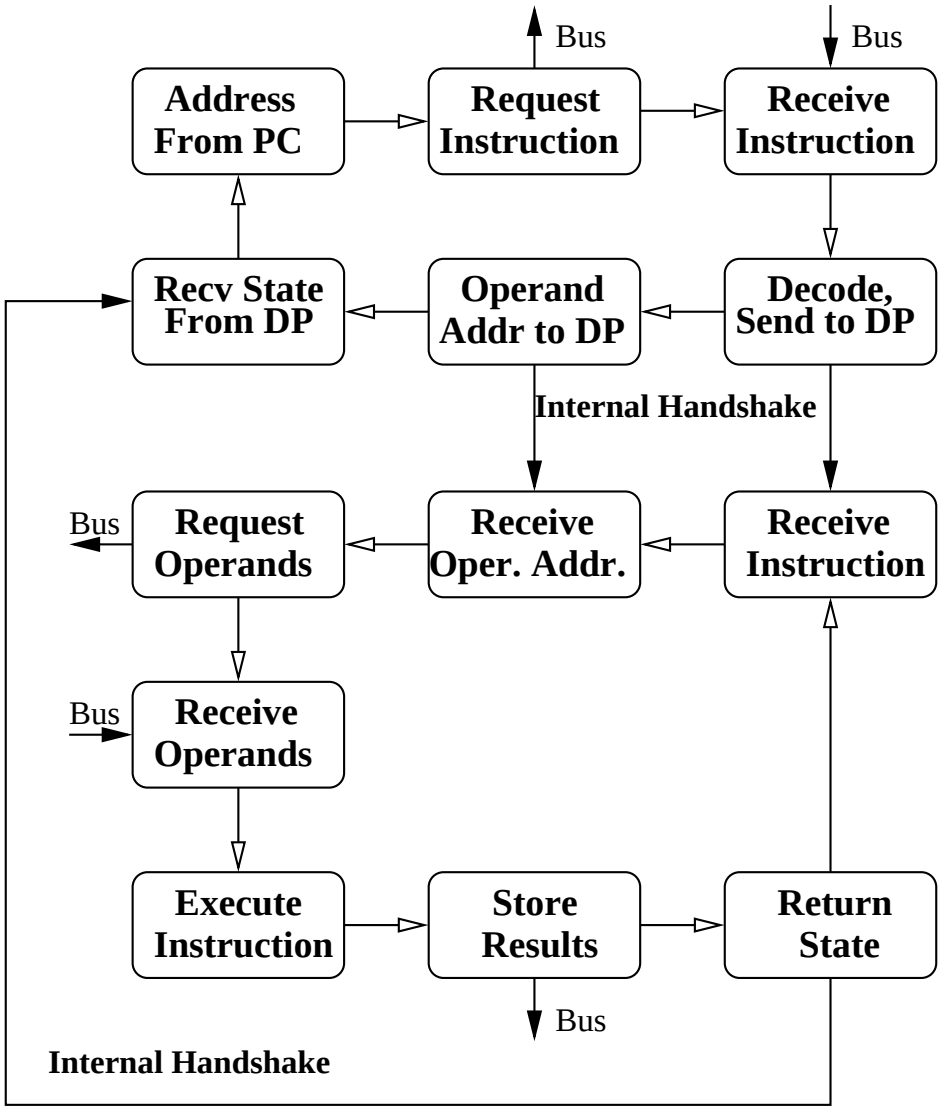
- Constants can be stored with Instructions in ROM.
- Better Bus balancing is possible.
- Typically, 1 instruction read, 1 constant read, 1 data read and 1 result write per instruction.
- 2 mem ops per bus.

Modified Harvard with Cache



- Cache allows optimum utilization of bus bandwidths.
- Each operation need not be balanced individually.

Instruction and Data State Machines



- Operation of the system may be modeled as two interacting state machines.
- Instruction processor fetches instr, decodes and gives operation type and operand locations to data processor.
- Data processor fetches operands, performs operation and writes back the result.

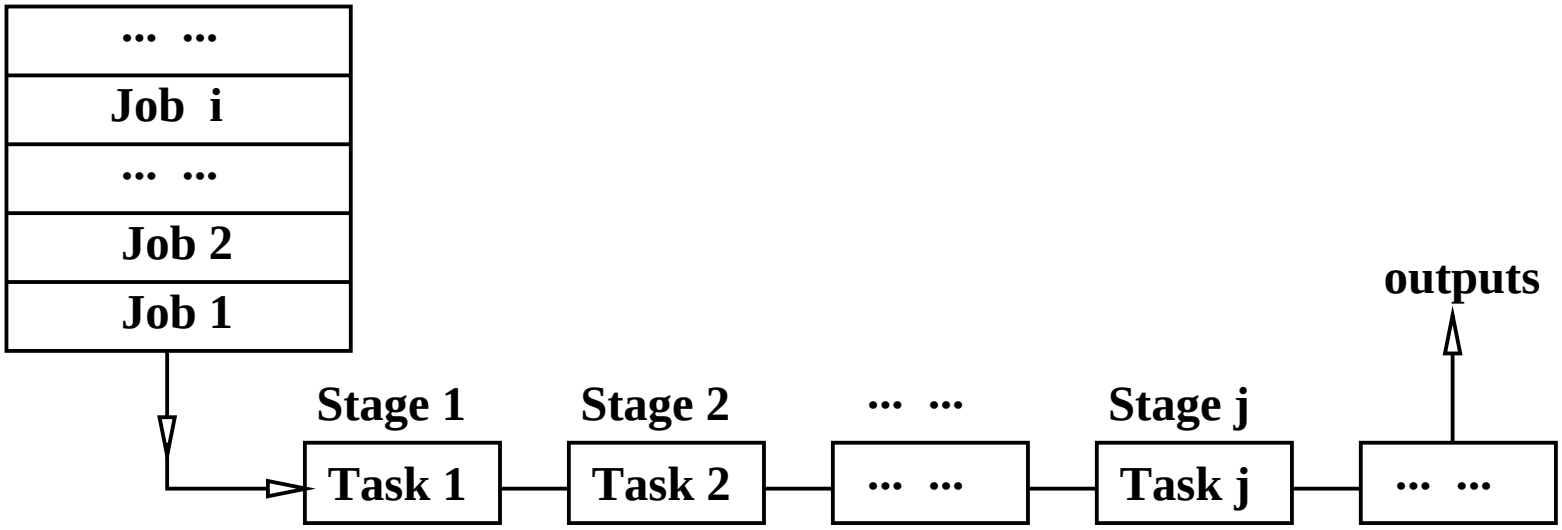
Lessons From Performance Evaluation

- Amdhal's law states that the performance improvement through any architectural modification is asymptotically limited to $1/f$, where f is fraction of *executed* instructions which are **not** improved by the modification.
- while deciding the architecture of a processor, we should prefer to spend our resource (silicon real estate, power . . .) on those instructions which are most frequently executed.
- One of the most frequently executed operations is memory access. This is also a slow operation when memory is external to the processor.
- By avoiding implementation of complex bur infrequently used instructions, we can use the silicon real estate for higher number of on chip registers.
- This suggests the use of a RISC architecture.

RISC architecture

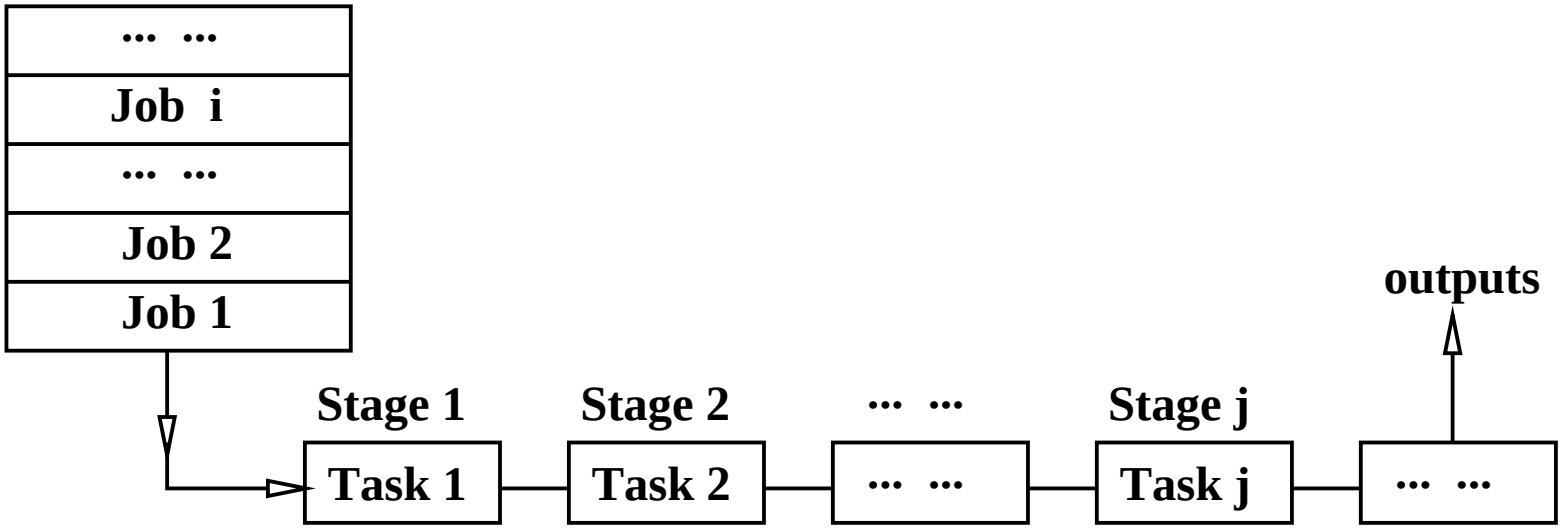
- Although by itself, the RISC architecture implies just a reduced and regular instruction set, which frees up silicon real estate for a larger number of registers and other useful hardware units like barrel shifters, most RISC processors have adopted other architectural features as well to improve performance.
- Most RISC processors are pipelined. In a pipelined architecture, the hardware works on several instructions at the same time. While one instruction is being fetched, the previous one is being decoded and the one before that is begin executed. This improves throughput.
- Most RISC processors also use a load-store architecture. This means that only the load and store instructions access the memory. All other instructions operate only on registers. If you need to operate on a memory operand, it should first be loaded into a register and then the operation is carried out.

Pipelining



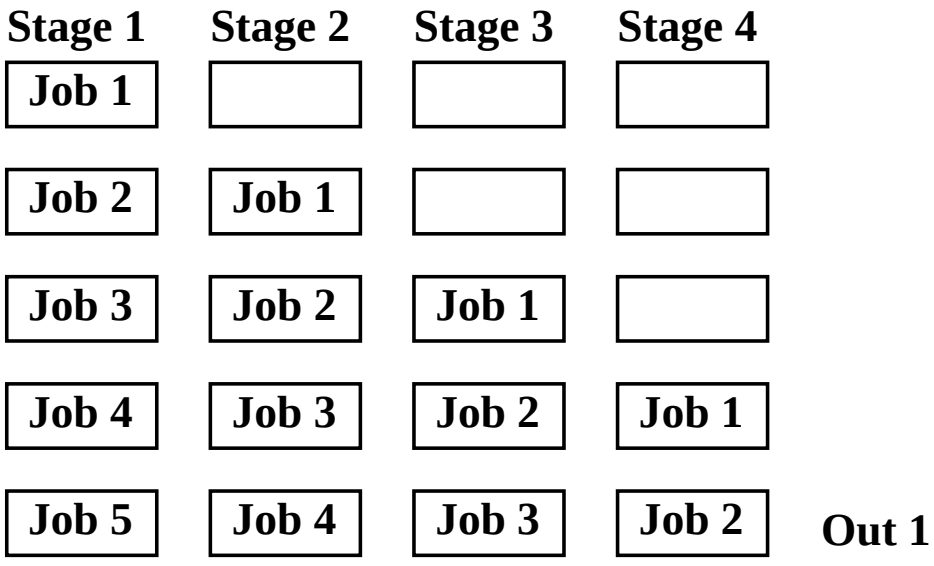
- We often have to perform several jobs repetitively, where each job requires many tasks to be performed in sequence.
- Each task may require specialised hardware which is specific to this task.
- A simple minded way to do this would be to take each job from the queue, perform all its tasks, then take up the next job and so on . . . .

Pipelining



- With the simple minded approach, only the hardware specific to the task being performed at a given time will be active, while all other stages will be idle.
- Can't we begin doing the next job using the idle hardware even before all the tasks for the first job are over?
- Indeed we can! This is called pipelining.

Pipelining



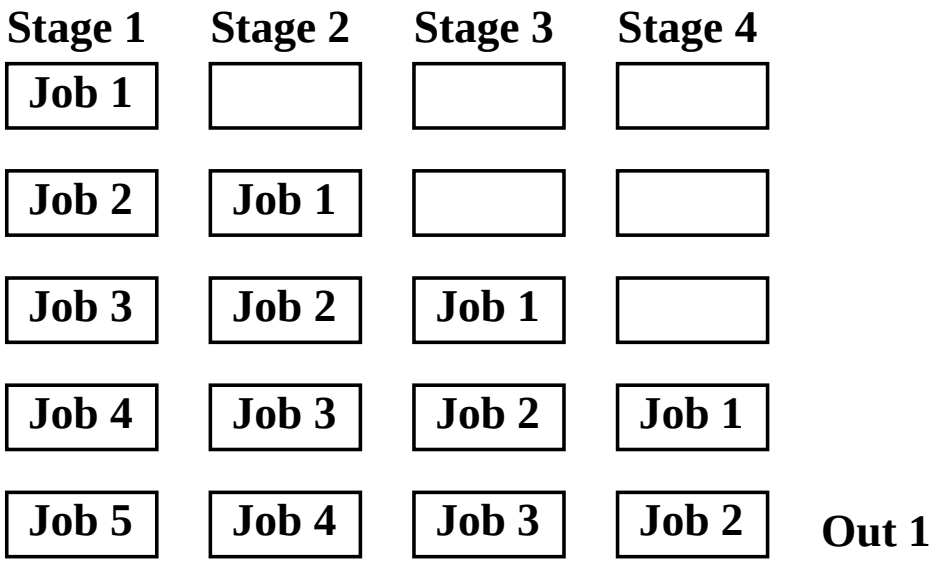
- Notice that the time between the arrival of a job and the availability of the *corresponding* output is not changed.
- However, new jobs can be taken up much more quickly and the outputs are generated much more frequently.

The time elapsed between the arrival of a job and the production of the corresponding output is called its latency.

The number of jobs handled per unit time is called throughput.

Pipelining improves the throughput of a repetitive process.

Pipelining



- Flow through the pipeline will be dominated by the slowest operation.
- For an optimal implementation, tasks should be so chosen so that all of them take the same amount of time.

In case of a processor, this suggests a constant sized instruction word and balanced operation time for the decode, execute and write-back stages.

Instruction Set Architecture

- The Instruction Set Architecture (ISA) is the programmer visible architecture of a processor, with hardware implementation details removed.
- We have seen the 8051 and the 8086 through this view.
- Notice that the same ISA may be implemented in different ways by hardware designers. Thus, the ISA lasts over several generations of hardware implementation.
- ISA is a convenient interface between the hardware designer and the programmer. The programmer can assess the performance of a processor without having to deal with hardware details. At the same time, the hardware designer can assess the importance of some features of the architecture and optimize these in the implementation.

The MIPS processor

MIPS is an acronym for Microprocessor without Interlocking Pipe Stages. We shall use it as an example of a RISC processor.

- For understanding its implementation, we shall use a simplified ISA.
- MIPS is a 32 bit RISC processor with a load-store ISA.
- One can see the adaptation of some of the ideas used in CISC processors in MIPS. For example, the equivalent of ACALL and AJMP instructions of 8051.
- It uses a 32 bit instruction word with a regular bit pattern to ease decoding and to speed up execution.
- The processor design permits the use of independent integer and floating point ALUs.

The MIPS processor: Registers

No.	Name	Function	Save Convention
$0	$zero	Zero value	N.A.
$1	$at	Assembler Res.	N.A.
$2-$3	$v0, $v1	Procedure results	N.A.
$4-$7	$a0-$a3	Procedure arguments	Must Save
$8-$15	$t0-$t7	Scratch Pad	Need not save
$16-$23	$s0-$s7	Operands	Must save
$24-$25	$t8-$t9	Scratch Pad	Need not save
$26-$27	$k0, $k1	For Kernel (OS) use	N.A.
$28	$gp	Global pointer	Must Save
$29	$sp	Stack pointer	Must Save
$30	$rp	Frame pointer	Must Save
$31	$ra	Return Address	Must Save

Register usage is by convention and not enforced in hardware.
The floating point execution unit has its own registers.

The MIPS processor: Memory

- MIPS uses a single address space for program, data and memory. (Von Neumann architecture).
- Multi byte numbers are stored in memory using the big-endian convention– most significant byte at the lowest address. This is different from Intel convention!
- Load and store operations must be *aligned* – a four byte quantity (word or instruction) must be loaded from or stored at an address divisible by 4, and a two byte quantity must be loaded from or stored at an address divisible by 2.
- Since all instructions are 32 bit wide, the two least significant bits of instruction addresses are always zero.
- In some instructions, these bits are not specified and are implicitly assumed to be zero.

The MIPS processor: Stack

- A stack is a storage area in memory which implements a last-in, first-out or LIFO data structure.
- Items are added to or removed from the stack using a pointer called the stack pointer. MIPS register $29 ($sp) is conventionally used as the stack pointer.
- A stack may grow towards higher addresses or towards lower addresses as items are added to it. correspondingly, it is called an Ascending or a Descending stack.
- The stack pointer may point to the last added item on the stack or to the first available empty location on the stack. Correspondingly, it is called a Full or an Empty stack.
- Thus there are four types of stack: Full Ascending, Full Descending, Empty ascending and Empty descending.
- MIPS uses a Full Descending stack (like 8086) by convention.

The MIPS processor: Instruction Format

MIPS uses a load-store architecture. Only load and store instructions access the memory. all others work with registers.

Instructions can have up to two source operands and a destination operand.

There are three instruction formats: Register type, Immediate type or Jump type.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
R	6 bits Op type	5 bits src1	5 bits src2	5 bits dst	5 bits shift	6 bits Operation
I	6 bits Op code	5 bits src/ Base	5 bits dst/ data	16bits Immediate/ Addr offset		
J	6 bits Op code	26bits Mem. word addr (addr bits 27-2)				

R type instructions

For all instruction formats, the top six bits decide which of the three formats is being used.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
R	6 bits Op type	5 bits src1	5 bits src2	5 bits dst	5 bits shift	6 bits Operation

- top 6 bits being zero indicates the R format and the bottom six bits identify the actual operation to be carried out.
- Except for some co-processor related instructions, all R type instructions have zeros as the top 6 bits.
- If the top six bits are 0100xx, the instruction is handled by a co-processor. The last two bits represent the co-processor number.

I type instructions

I type instruction format is used when we need to include an immediate operand or when we use register relative addressing for an operand.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
I	6 bits Op code	5 bits src/ Base	5 bits dst/ data	16bits Immediate/ Addr offset		

- Since the lower half word is used up by the immediate value or the address offset, the top six bits must encode the operation to be carried out.
- In case of load/store operations, the data register is the destination or the source of load/store operation. Contents of the base register and sign extended addr. offset are added to get the memory address.

J type instructions

J type instructions are used for branching and function calls.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
J	6 bits Op code	26bits Mem. word addr (addr bits 27-2)				

- Since all instructions begin on addresses which are multiples of 4, the bottom two bits are always zero and need not be specified. The address with the bottom two bits removed (=address/4) is called the word address.
- The 32 bit destination address of jump/call is composed by taking the top 4 bits from the program counter, next 26 bits from the instruction and appending two zeros as least significant bits. (Similar to 8051 AJMP/ACALL).
- Top six bits define the actual operation.

MIPS Processor Instructions and Pseudo Instructions

- Since MIPS is a reduced instruction set processor, it implements a minimal set of simple instructions.
- Many common operations then need to be performed using multiple processor instructions. (For example, a push on stack is done by using separate processor instructions for stack pointer update and for storing in memory).
- To reduce the burden on the programmer, MIPS assemblers define many pseudo-instructions. These are translated by the assembler to one or more actual MIPS implemented instructions.
- In fact, a register ($1 or $at) is conventionally reserved for the assembler in order to compose useful pseudo-instructions and addressing modes not supported by MIPS processor as single instructions.

Arithmetic and Logical instructions

- MIPS implements signed and unsigned addition, subtraction, multiplication and division of register contents through R type instructions.
- It also implements logical instructions like bit-wise AND, OR, XOR and NOR of register contents through R type instructions.
- MIPS support addition, bit-wise AND, and bit-wise OR with immediate constants through I type instructions.
- It also supports left shift, logical right shift and arithmetic right shift through R type instructions which can include a 5 bit shift amount.
- Many operations (for example subtraction of immediate values from a register) are implemented as pseudo-instructions by the assembler.

Addition and Subtraction

- Addition and subtraction of register operands is supported by R type instructions add and sub. Unsigned versions of these are carried out through R type instructions addu and subu.

  Addition/subtraction process is identical for signed and unsigned operations – however, an overflow condition will cause an interrupt in case of signed operations and will be ignored in case of unsigned operations.
- I type instructions addi and addiu operate the same way as register operations, except that these add a sign extended version of a 16 bit constant contained in the instruction to a source register.

  Subtract immediate is implemented as a pseudo-instruction by the assembler by taking the negative of the constant value and adding it.

Multiplication and Division

- Signed and unsigned multiplication are also implemented through R type instructions.
- Multiplication of word sized operands produces a double word output, while division of a word sized operand by another produces a word sized quotient and a word sized remainder – so the result is 2 words in both cases.
- Result of 2 words is managed by having dedicated word sized registers Hi and Lo associated with multiplication and division.
- High word of result after multiplication is place in Hi, while the low word is placed in Lo. Similarly after division, the quotient is placed in Lo, while the remainder goes to Hi. (Similar to 8086 with AX-DX).
- There are specific instructions for moving data between general purpose registers and Hi/Lo.

Multiplication and Division

- Instructions for signed/unsigned multiplication and division are mult, multu, div, divu. All of these are R type instructions.
- Results of these operations go to Hi and Lo registers, which are a part of the multiply-divide hardware in the ALU.
- Data movement from/to the Hi register is provided by mfhi (move from hi) and mthi (move to hi) instructions.
- Data movement from/to the Lo register is similarly provided by mflo and mtlo instructions.
- Multiplication and Division by immediate values is not implemented by MIPS.
- Assemblers provide pseudo-instructions combining mult with mfhi and mflo, such that the results end up in general purpose registers.

Logic Functions

- MIPS implements bit-wise logic functions between registers. These R type instructions are: AND, OR, XOR and NOR. All of these are R type instructions and specify the two source registers on which the bit-wise logic operation is carried out and the destination register.
- MIPS also supports bit-wise logic with an immediate value, but only AND and OR functions are supported with an immediate operand.
  These instruction use the I format, of course.
- Immediate versions can be implemented easily as pseudo-instructions by putting the immediate in the assembler reserved register and then carrying out register to register operation.
  (Some implementations support XOR with immediate as a native instruction xori).

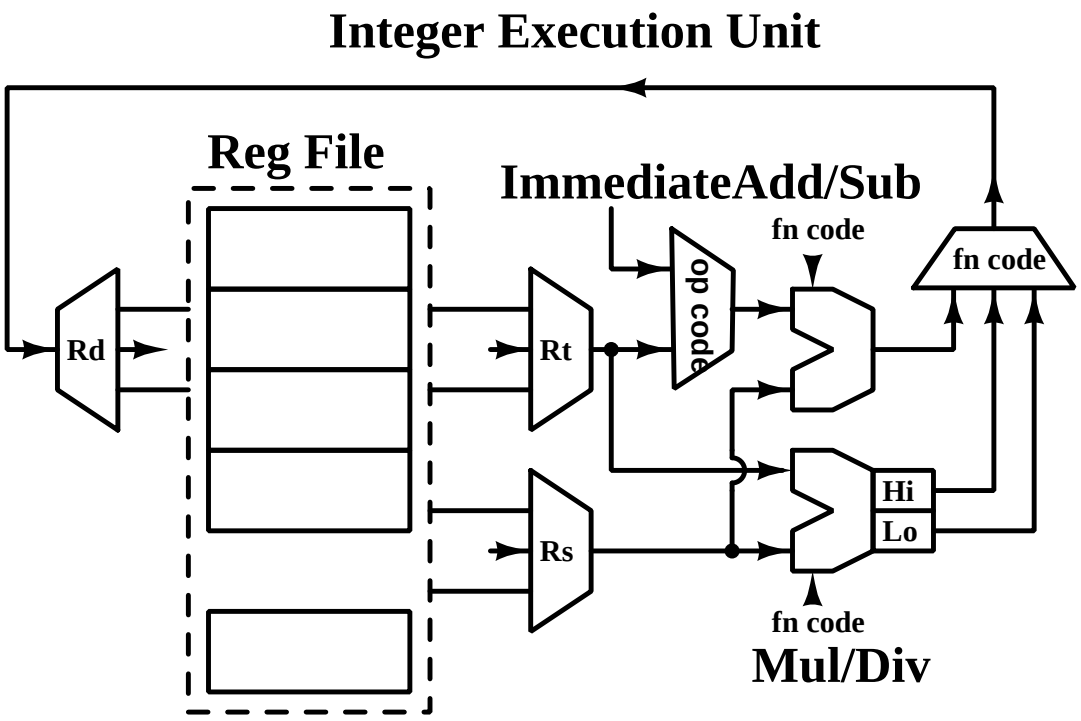
Shift operations

- MIPS implements shift left, shift right logical and shift right arithmetic as native operations.
- The instructions are R type and specify the operand in one register and the shift amount either as a 5 bit immediate value or in another register.
- The mnemonics for the instructions with immediate shift values are:
  sll dest, src1, shamount; srl dest, src1, shamount
  and sra dest, src1, shamount.
  Notice that these instructions use the R format, which has a field for shift amount.
- When the shift value is in another register, the mnemonics are sllv, srlv and srav. (Shift left logical variable . . .). Here src1 has the operand and src2 contains the shift amount.

Comparison operations

- MIPS does not have a flag register. Comparison instructions allow one to store the result of a comparison into a destination register.
- The register comparison instruction is slt (set if less than). This is an R type instruction and is used as slt rd, rs, rt. Register rd is set to '1' if rs $<$ rt, else it is set to zero.
- To compare with an immediate number, we use the instruction slti. This is an I type instruction and is used as slti, rd, rs, value
  Register rd is set to 1 if rs $<$ value, else it is set to zero.
- Comparison instructions are typically followed by conditional branching.

MIPS ALU: Integer Unit



- The register file provides two output ports and an input port.
- Multiplexers controlled by Rs/Rt fields in R or I format instructions connect chosen registers to the appropriate function units (add/sub or mult/div).

A demultiplexer controlled by the Rd field in the instruction connects the appropriate output to the chosen destination register.

Data Movement

Register to Register Interestingly, the MIPS processor does not have a register to register move instruction for general purpose registers.
We achieve the same result by adding zero to the source register and storing the "sum" into the destination register.

Since move is such a common instruction, most MIPS assemblers provide a pseudo-instruction called move, which is translated by the assembler to the addition based implementation.

Assembler	Processor	Effect
move $t2, $t1	add $t2, $zero, $t1	($t1) $\rightarrow$ ($t2)

We have already seen the special purpose instructions to move data to and from Hi and Lo registers.

Data Movement

Immediate value to Register Immediate values are constants contained within the instruction itself.

Since all instructions are 32 bit wide, this presents a problem – we cannot accommodate a 32 bit wide immediate value into the instruction. The I type instruction format does have room for 16 bit immediate values.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
I	6 bits Op code	5 bits src/ Base	5 bits dst/ data	16bits Immediate/ Addr offset		

We can use instructions using this format to load halves of a register with immediate values.

The add immediate (addi) instruction uses I format and can be used for adding $zero and a 16 bit immediate value and putting the result into the lower half of a register.

Data Movement

Since move immediate is a common operation, assemblers provide a pseudo-instruction – load immediate (li) to place an immediate value into the lower half of a register.

Assembler	Processor	Effect
li $t3, val16bit	addi $t3, $zero, val16bit	$t3 = val16bit

Thus the RISC processor can dispense with implementation of instructions like register mov and move immediate.

The corresponding actions can be provided by other instructions and the assembler can provide pseudo-instructions to translate the more conventional forms of these statements to the ones which are implemented by the RISC processor.

The MIPS processor: Data Movement

How can we get an immediate value into the upper half of a register?

Immediate value to upper half of register The processor provides an instruction using the I format for loading a 16 bit value into the upper half of a register.

The instruction is called lui or load upper immediate.

Notice that the 'u' here stands for upper and not for unsigned. (There is no question of sign extension when the value is being loaded in the upper half).

For example,

lui $t4, val16bit; will load val16bit in the upper half of $t4 and clear its lower half.

Data Movement

32 bit Immediate value to a register For this, we first load the upper half using lui and then OR the lower half with a 16 bit immediate value using ori.

Again the assembler pseudo-instruction load immediate (li) is used for this. If the immediate value is 32bit wide, it is broken up into upper16bit and lower16bit. The pseudo-instruction li is now replaced by two processor instructions: lui and ori.

Assembler	Processor	Effect
li $t5, val32bit	lui $t5, upper16bit;	Load upper half
	ori $t5, lower16bit;	merge lower half

The final effect is that the 32 bit value ends up in the register in the correct position.

Data Movement

Clearing a register A special case of load immediate is when we want to initialize a register to all zero's. We do have a register ($zero) which has all zeros in it. However, we cannot just copy it to another register – since there is no mov instruction implemented by MIPS. We must replicate it by addition to itself.

Again, the assembler provides a pseudo instruction which is translated to the appropriate action that will load all zeros in the desired register.

Assembler	Processor	Effect
clear $t6	add $t6, $zero, $zero	$t6 = 0

We can also xor a register with itself to clear it.

Data Movement to and from memory

Load and store operations are the only ones that access data from the memory. These use the I format instruction, adding the base register and addr offset to get the memory address. The data register then becomes the destination/source for load/store.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
I	6 bits Op code	5 bits src/ Base	5 bits dst/ data	16bits Immediate/ Addr offset		

- Instructions lw, lh and lb load a word/half-word/byte from memory into a specified register from a memory location. If a half-word or byte is loaded, it is sign extended to fill the destination register.
- If we don't want sign extension, instructions lwu/lhu/lbu should be used.

Data Movement to and from memory

- We can store a word/half-word/byte to memory using instructions sw/sh/sb.
- In this case there is no difference between signed and unsigned, since there is no size extension in memory – a byte occupies a single byte and a half word occupies two bytes in memory.
- The standard addressing mode using I type native instructions for load/store is base+offset. However, other addressing modes can be constructed by pseudo-instructions.
- For example, we can add a base register to an index register using the addu instruction, storing the sum in the assembler reserved register $at.
  Now using $at as the “base” register in load/store native instructions, we can construct based-indexed addressing modes.

Jumps and Calls

The MIPS ISA provides

- Jumps to labels within a 256 MB window around the updated PC,
- Jumps to an absolute address contained in a register,
- Conditional jumps to addresses relative to PC,
- Function calls analogous to the above, where in addition to a jump, the return address is copied to the special function pointer register $ra.
- Functions then return using a register jump with the target address being the content of $ra.

Jumps and Calls

Jump to program lables use the J type instruction format.

Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6
J	6 bits Op code	26bits Mem. word addr (addr bits 27-2)				

- The instruction for jump is simply ‘j label’. The label must lie within a 256 MB window around the current instruction address. (It should have the same top 4 bits as pc).
- The actual destination address is constructed by concatenating the 4 top bits from pc, the 26 bits given in the instruction and two zeros as the least significant bits to construct the 32 bit address.
(Similar to AJMP of 8051).
- The corresponding function call instruction is jal – jump and link. Here, the target address for the jump is constructed as above. Additionally, the updated pc (the return address) is written to the $ra register.

Jumps and Calls

What if we want to jump to a location which does not have the same 4 top bits as the updated pc?

- The instruction ‘jr $reg’ takes the destination address from a specified register and copies its 32 bits to pc.
- This provides unconditional jumps spanning the full address range.
- The corresponding function call instruction is ‘jalr $reg’.

The called function returns by doing a jr using the register $ra.

- If the called function calls yet another function, $ra will be overwritten. To avoid losing the return address, its value must be saved.
- Since the depth of calling cannot be limited to some number, we need a growing data structure like a stack to save it.

Conditional branching

- All conditional branching instructions use a pc relative address to branch to. Thus the base address for the destination address is implied (PC).
- I type format is used for these instruction. So the 16 bit signed offset from PC is included in the instructions.
- Conditions for executing the branch depend on the two register fields.

MIPS provides the following conditional branching instructions:

beq	reg1, reg2, label	Jump to label if (reg1) = (reg2)
bne	reg1, reg2, label	Jump to label if (reg1) $\neq$ (reg2)
bgez	reg1, label	Jump to label if (reg1) ≥ 0
bgtz	reg1, label	Jump to label if (reg1) > 0
blez	reg1, label	Jump to label if (reg1) ≤ 0
bltz	reg1, label	Jump to label if (reg1) < 0

Conditional branching

beq	reg1, reg2, label	Jump to label if (reg1) = (reg2)
bne	reg1, reg2, label	Jump to label if (reg1) $\neq$ (reg2)
bgez	reg1, label	Jump to label if (reg1) ≥ 0
bgtz	reg1, label	Jump to label if (reg1) > 0
blez	reg1, label	Jump to label if (reg1) ≤ 0
bltz	reg1, label	Jump to label if (reg1) < 0

- Notice that only equality and inequality can be tested between two registers.
- For other comparisons, one of the registers is implied to be zero.
- But the I format has room for two registers. So why is this restriction placed?

Conditional branching

- The reason for restricting greater/less than type of comparisons to the implied register $zero is that none of the conditional branches involve the adder/subtractor.
- Equality can be tested between two registers without needing the adder.
- Other comparisons are possible with zero without using the adder.

But what do we do if we need greater/less than type comparisons between two registers?

Conditional branching

- What do we do if we need greater/less than type comparisons between two registers?
- We first use slt or sltu instructions, which use the adder. The format is:
  slt/sltu dest, src1, src2; (u form is used for unsigned comparison)
- This sets the destination register to ‘1’ or ‘0’ depending on whether the result of comparison is false or true.
- This can now be followed up with a conditional jump like beq or bne using comparisons of the destination register of slt with the $zero register.
- The assembler uses this technique, storing the result of slt in $at. Using this, it provides pseudo-instructions:
  bge, bgeu, ble, bleu for conditional branches depending on comparison between two registers.

Branch with link

- MIPS supports versions of unconditional and conditional jumps which store the return address in the special function register $ra before taking the jump.
- These are used for calling functions.
- The called function returns by executing jr $ra.
- If the function is going to call another function, it should first save $ra somewhere – typically on the stack.
- Jump and link instructions are:
  jal; jalr; bgezal, bgtzal,bltzal.
- These correspond to j, jr, bgez, bgtz and bltz.