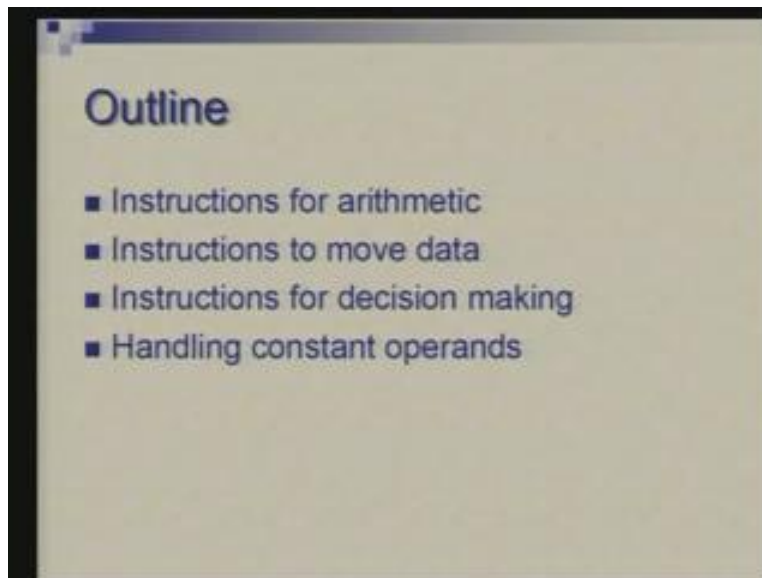


Computer Architecture
Prof. Anshul Kumar
Department of Computer Science and Engineering,
Indian Institute of Technology, Delhi
Lecture - 3
Instruction Set
Architecture - 1

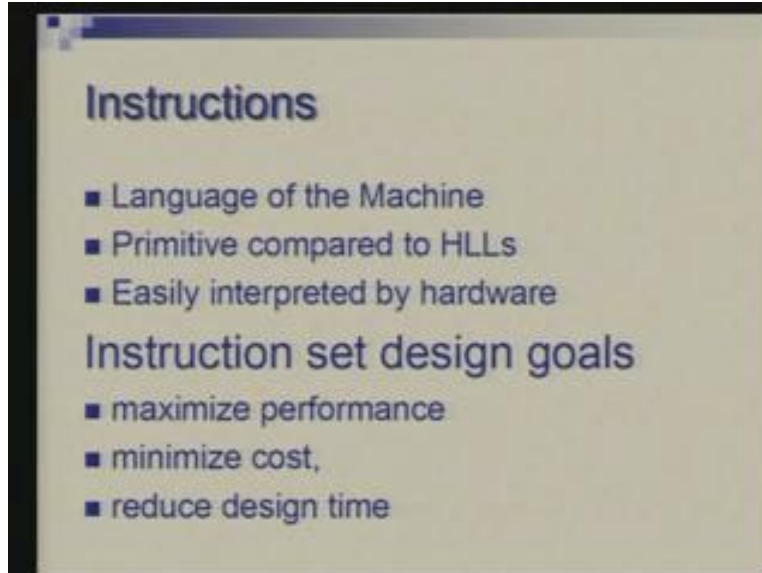
Today I will start discussion on instruction set architecture. As I discussed earlier in the first and second lecture, instructions forms the interface between hardware and software, instructions provide the primitive operations in terms of which computation has to be described and from the hardware point of view instructions are the basic behavior definitions which hardware has to implement. So what we will do today is to take very simple instructions, try to understand what action they perform and also where in a program they can be used to do useful things.

(Refer Slide Time: 1:40)



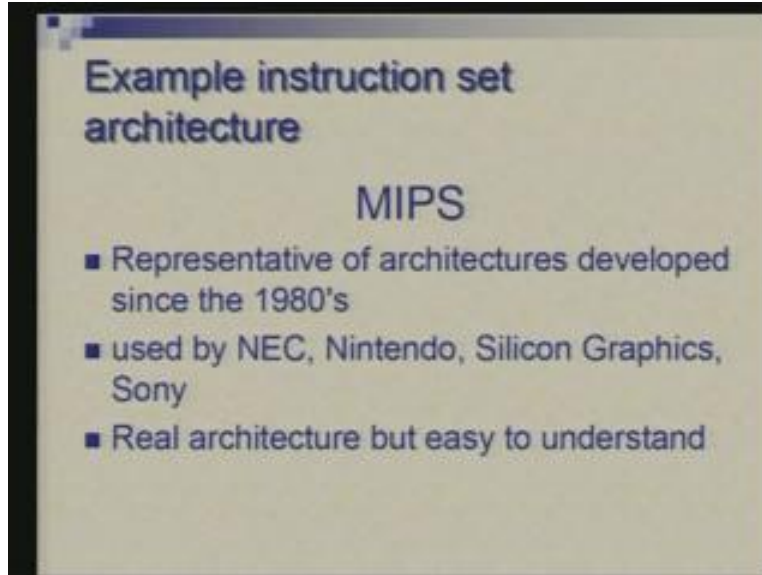
First of all we will look at some instructions for carrying out arithmetic operations simple ones such as addition, subtraction then we will see how instructions are used to move data, the data is required to be moved for example between memory and registers then we will see how decisions are made in program so how flow of control is defined using instructions and then we will take special care of how to handle constant operands. There are many situations where constant operands which have to participate in computations. These are the simple instructions which are almost used in all situations and in almost all programs.

(Refer Slide Time: 2:27)



Now we will have to carry the distinction between assembly and machine instructions. So we will be talking of the relationship between machine language and assembly language. Machine language is basically the most basic building blocks for any program and it is something which hardware can interpret. so when you are designing an instruction set when you are designing a new architecture you have to have certain goal; the goal is to provide a set of operations where computation can be expressed efficiently so you need to maximize performance, maximize efficiency and at the same time the cost of implementing these instructions hardware should be as little as possible. Also there could be other goals as power consumption and so on and it should be simple so that design time is possible.

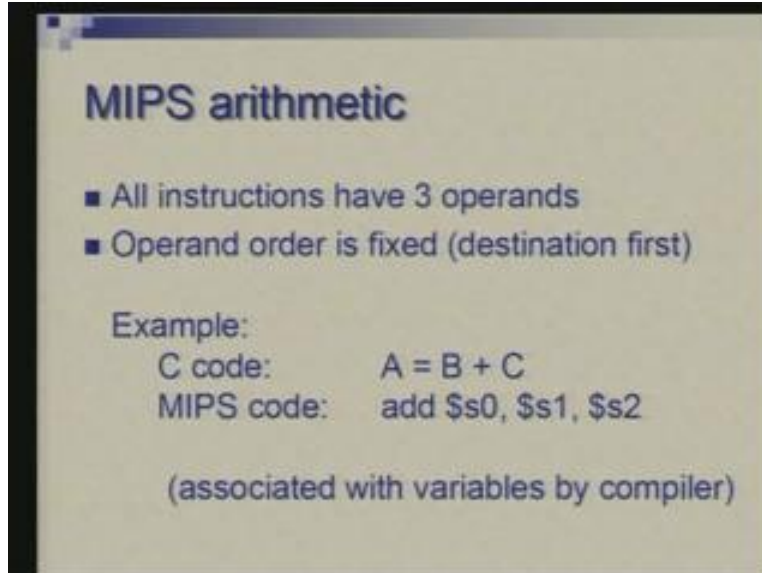
(Refer Slide Time: 3:32)



Now when you talk of an instruction set for the purpose of reaching you have to always start with an example. Often one takes a toy example a toy machine and describes that. But on the other hand, we will take real machine but a simple one in the series which is called MIPS this was developed by people at Stanford in early 80s. This is a typical architecture which is come to known as risk reduced instruction set computers. We will talk more about that later but one thing which is important is that from 80s onwards these are the typical architectures which have been developed so subsequently there have been all developments along these lines.

MIPS in particular is used in several applications; not all of them are general purpose computing. Example you can see here are: NEC, Nintendo which is video games, silicon graphics computers and also Sony play station. So it is a real architecture that is the plus point. We are talking of a toy architecture but at the same time it is a fairly simple one to understand and within a short period you will get a full grasp over this simple architecture.

(Refer Slide Time: 5:13)



MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

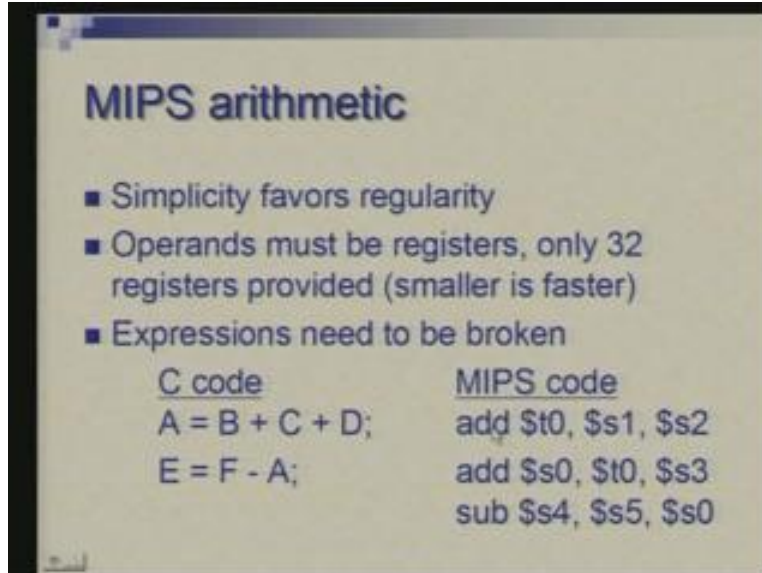
C code:	A = B + C
MIPS code:	add \$s0, \$s1, \$s2

(associated with variables by compiler)

We begin with arithmetic instructions. The simplest arithmetic operations has been add and subtract and in simplest form you need to add two numbers to produce the result. So there are three things to be specified with the instructions as which are the operands and where is the destination of the result. Suppose in C you have an assignment like: A is assigned B plus C sum of B and C then equivalently in MIPS assembly language we will write add (Refer Slide Time: 5:52) here is a register name dollar s0, dollar s1 another register, dollar s2 another register. So this dollar is just to specify just to signify that it is a register name it is a special symbol s0 and the association between registers like this and the variable names like A B C would be established by compiler. for example, the compiler comes across a statement like A gets B plus C it would translate this into an add instruction and at the same time figure out where B is going to be sitting, where C is going to be sitting and where is A going to be replaced. We have taken arbitrarily some decision that these are in s0, s1 and s2 these are some registers of MIPS machine.

Now one thing you would notice here is the order in which s0, s1, s3 are appearing in the instruction is very significant. unlike the C code where you are saying A assign B plus C by this infix symbols it is very clear which are the opponent where is the destination but here it is just appearing as a list. So it is clear by convention only that the first one is the destination and the next two are operands so we are adding contents of s1 register, s2 register and putting the result in s0 register.

(Refer Slide Time: 7:38)



MIPS arithmetic

- Simplicity favors regularity
- Operands must be registers, only 32 registers provided (smaller is faster)
- Expressions need to be broken

<u>C code</u>	<u>MIPS code</u>
A = B + C + D;	add \$t0, \$s1, \$s2
E = F - A;	add \$s0, \$t0, \$s3
	sub \$s4, \$s5, \$s0

Now, this is simple and in fact simplicity has been one of the design goals for this architecture and simplicity favors regularity. What it means is that you would try to follow certain uniformity, various operations, for example, if you take subtract instruction this will also follow similar format. And also if you think of addition of three numbers or four numbers or five numbers there are no separate instructions for doing so, so you have to define all those in terms of the same primitive instruction which adds two numbers.

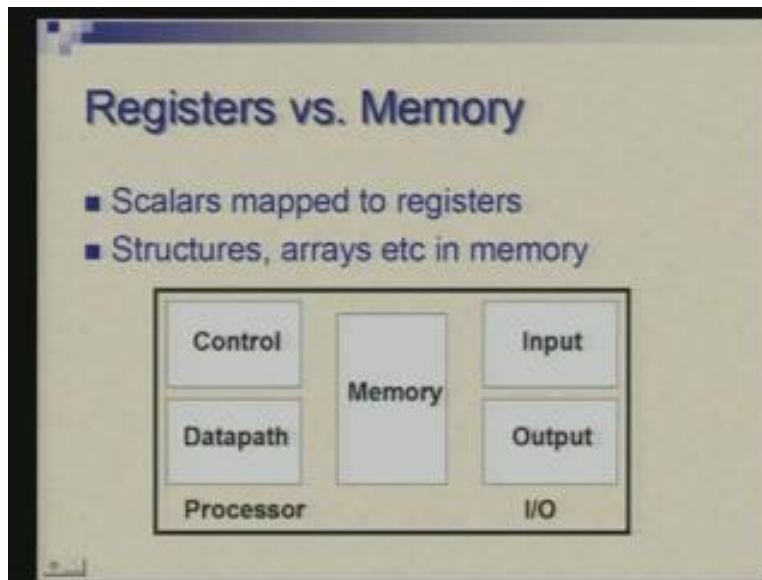
Now the operands in MIPS are 32-bit numbers and therefore registers are 32-bit registers. Question here would be that why we are limiting, why not say that add instruction can add two numbers but they could be of arbitrary size. Once again you have to limit the size because working on arbitrary size would not necessarily give you the efficiency. If most commonly your numbers can fit within 32 then it is good to have a limit like this and at least make sure that addition of 32-bit numbers is fast and can be done in a single step. If you want to add larger numbers then occasionally whenever you need that you can go through a sequence of instructions and possibly spend more time but the common case can be done at a fast speed.

Therefore, when you have complex expressions such as addition of three numbers or something which involve several operations then it has to be broken down into a sequence. Now look at these two statements (Refer Slide Time: 9:47) A gets B plus C plus D and another statement which subtracts A from F which is the result of the previous instruction. This could be written as a sequence where you have two additions being done. We are assuming here that the value of C is present in s one, D is present in s2 the result is going to t0 which is a temporary location; it does not correspond to anything directly in the high level language program.

What did I say; did I say C and D or B and C? It will be B and C which will be added and then t0 contains the sum of B and C and D which is the third operand in the first

statement in s3 and the result now of these two is A so s0 is the seat of A. For the second instruction we have to be now careful that it is this value of A which has to be used in the second instruction so you find that s0 is there as the second operand in the subtraction operation. This is the restate forward. Once you know how to do primitive operations anything which can be done by putting these together can be done in straightforward manner.

(Refer Slide Time: 11:22)



Now if you have everything in register what is memory for?

Of course memory is there to hold the program but memory also is used to hold bulk data, larger data structures for example: records, structures, arrays or other complexing you build they have to be in memory which can have much more capacity. The number of registers is limited. In MIPS particularly there are 32 registers so basically you can have only a few scalars which can be mapped to the MIPS registers. But when you are talking of arrays then they have to be kept in memory which means you need to have instructions to move data between memory and registers so that is what we will see next. But before we go for that we need to see how we get data from specific locations of memory.

(Refer Slide Time: 12:04)

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data

Memory you could view as a large one dimensional array consisting of bytes. This is again a convention that each addressable unit in the memory is considered to be a byte whereas operands I am talking of are 32-bits which basically means that mostly you have 32-bit operations but sometime you may need to look at half the word of 32-bits or one quarter so the addressability is provided at a final resolution that means you can address each individual byte. So each word will have four bytes and each byte can be addressed so addresses are all in terms of byte addresses.

Memory address is nothing but an index into this array and this specify the byte number.

(Refer Slide Time: 13:22)

Words and Bytes

- 2^{32} bytes : byte addresses from 0 to $2^{32}-1$
- 2^{30} words : byte addresses 0, 4, 8, ... $2^{32}-4$

	0	1	2	3
Little endian byte order	4	5	6	7
Big endian byte order	3	2	1	0
	7	6	5	4
Non-aligned word	3	2	1	0
	7	6	5	4

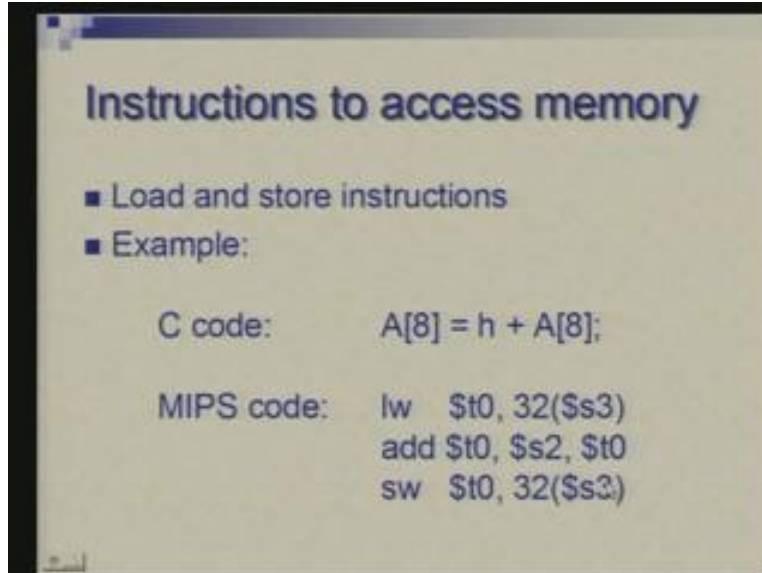
So with 32-bits of address you can specify 2^{32} bytes and these are addressed from 0 to $2^{32} - 1$. If you are talking of words 32-bit words each consisting of 4 bytes these will have typically addresses 0, 4, 8 and so on that is multiple of 4 and therefore you have a total of 2^{30} words which go from 0 onwards and the last one would be $2^{32} - 4$.

What is the relationship of bytes with word?

There are two possibilities here. These are two different conventions which are called... well just ignore this spelling error little endian or big endian. Little endian means that within a word you start numbering bytes from..... just a minute; I think these two are reversed this is actually big endian (Refer Slide Time: 14:28); you are starting numbering from the most significant side. So, on the left side I am showing most significant bit of a word and the right side there is least significant bit of the word. This is a big endian convention (Refer Slide Time: 14:40) whereas this is little endian convention. So you are starting from little end or the LSB Least Significant Bit end.

Different machines follow different conventions. For example; Intel processors follow little endian convention, spark processor follows big endian convention and when you are going to work in a lab with a simulator depending upon which machine you are running it on you will see different conventions so a simulator in particular adapts the convention of the host machine. So, if you are running on Pentiums you will find one convention, if you are running on CC 750 in Computer Centre you will find different conventions. This is as far as when the words are aligned with an address which is a multiple of 4. But there may be also situations where words are not aligned to address which is multiple of 4. So this shows an example at the bottom (Refer Slide Time: 15:55). So here we have a word beginning with this byte, this is the next byte, this is the next byte and this is the fourth byte. This is a logical word. Logically it is beginning at this point and ending at this point. Physically these bytes are grouped as shown here. Physically in the memory it is one word which consists of byte 0 1 2 and 3, another word which consists of byte 4 5 6 and 7 but your program logically can pick up a word from byte address 1. What it means is that you are looking at a word consisting of byte 1, 2, 3 and 4 so this logical word is spread over two physical words.

(Refer Slide Time: 16:41)



Now let us come back to instructions with excess memory which allows data to be moved between registers and memory in other direction. The two instructions are load and store. Suppose you want to pick up an element of an array, add something to that and store it back. A is an array and you want 8th element let us assume that it is an integer array. So here is the load instruction which is loading something from memory, one word from a memory specified by this address into a register t0. So here is lw where actually l stands for load and w stands for word. We are loading a word there are instructions for load, byte and so on but let us not worry about that right now. So the address is being specified in two parts; I am writing a number 32 and a register s3.

So at this point I am imagining that register s3 holds the starting address of the array and this offset or the index 8 we are saying eight elements of type integer which means that there is a byte offset of 32. So the sum of these two numbers: one number is 32, another number which is contained in register s3 these two together get added and define the address. So a word from this address is loaded into t0 and next we perform an addition corresponding to this add so t0 now contains the data loaded from memory. We are assuming that s2 contains the value h and the result is put in register t0 and then next instruction writes this value t0 into address specified by this. So this is store instruction and sw stands for store word (Refer Slide Time: 19:06).

The address here for store is same as load so basically we have put it in A[8]. Suppose you have to put it in A[10] then you would change this constant. Now you may be wondering what happens if it is something like A[i] we will worry about that later. Now I have described a simple situation where you can have constant indices into an array. There could be other complex situations. For example; the index of array could be a complex expression itself we will look at that later. But in a very simple situation we have seen how we can get data from memory, perform arithmetic and put the result back in memory.

(Refer Slide Time: 19:56)

A simple example

■ What does this code do?

```
swap(int v[], int k);
{ int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

→

```
swap:
  muli $2, $5, 4
  add $2, $4, $2
  lw $15, 0($2)
  lw $16, 4($2)
  sw $16, 0($2)
  sw $15, 4($2)
  jr $31
```

Now here is another example which tries to be little more. There is no arithmetic involved here it is simply a matter of moving data. What we are trying to do is trying to interchange two elements of an array and here you will see an example of a variable index. It is written in the form of a small function which is taking an array and an index as two arguments; it uses a temporary variable to interchange these two. It is a standard thing you do in programming; for interchanging two variables or two elements of some structural view go through a third temporary and cycle them like this.

Now this will involve basically....(Refer Slide Time: 20:57) it is clear that you require two loads where you will get these two $v[k]$ and $v[k+1]$ and two stores where you put values back in $v[k]$ and $v[k+1]$. The first two instructions are actually preparing to get the right address. This is another instruction which I have not talked of, it is a multiply instruction so it is trying to multiply contents of register 5 with 4; I am using 4 as a constant here I am not saying register 4 it is not dollar 4 it is just 4. I will elaborate on these constant operands little later. But here just interpret this as 'multiply instruction' it takes one value in five which is in this case k and it is multiplied by 4. Now this multiplication is being done because k is indexing into an integer array and for a byte offset we require four times that. So what we have in register 2 as a result of this multiplication is four times k that is added to the starting address of array $[v]$ which is contained in 4 to begin with and the final address is prepared in register 2.

Now you can load from this memory address which is zero offset 0 offset and the variable part in register 2. So, because we have done entire address calculation by these two instructions the complete address is in register 2 and offset is 0. So, load a word from here into register 15, the next word you pick up from a same address with four offset so basically you will pick up $v[k+1]$. This is effectively loading $v[k]$ in register 15 and the second load loads $v[k+1]$ into register 16 and all that we are doing is now while storing we just store in opposite order that is 16 is stored with zero offset and 15 is stored

with four offset so the two get interchanged so essentially we have not explicitly used temporary or another way of looking at this is we have used two temporaries we loaded both and then we have put in reverse order.

Ignore this for the moment (Refer Slide Time: 23:56) this is basically there because this is written as a function and this is like a return again we will talk of this later.

(Refer Slide Time: 24:08)

Machine Language

- Instructions are 32 bits long
- registers have numbers 0 .. 31, e.g., \$t0=8, \$t1=9, \$s0=16, \$s1=17 etc.
- Instruction Format:

Example: add \$t0, \$s1, \$s2

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

Now all along I was talking of instructions written in symbolic form so basically I was writing not quite in machine language I was writing in assembly language.

How do machine instruction or machine language looks like?

The instructions in binary form are 32-bit long so again there is uniformity, the data is typically or most commonly 32-bit long and the instructions are also 32-bit long. Now within an instruction, for example, if you take add instruction we need to say that the operation is add operation and there are three registers involved. Registers themselves are numbered from 0 to 31 because we have a total of 32 registers. So, for example, registers t0 t1 etc are numbered 8, 9 and so on; registers s0, s1 etc are numbered 16, 17 and so on so you might be wondering about the registers before 8 or after 23, we will again talk of that later.

But as we have seen in previous examples we have been using these registers labeled as t0 t1 and so on or sometimes we also use registers directly by numbers like register 2, register 5, register 15 and so on that is one way you can address and the other way you can have is in symbolic form t0, t1, t2 or s0, s1, s2. So now taking this example add dollar t0 dollar s1 dollar s2 how do we represent this in binary form.

So a 32-bit word is divided into several fields each field specifies a different part of the instruction. So in this case we have six fields. This is a six bit field (Refer Slide Time:

26:06) 6 5 5 5 5 and 6 the two at the end are 6-bit fields and all in the middle are 5-bit fields. The first bit is called the opcode field or op in short. This specifies which operation is being done, which operation the instruction is asking the hardware to do. So the code the opcode for add is all zeros. Subtract will have a different code, multiply will have a different code, divide will have a different code so this signifies what the instruction is about. But this being only six bits it will actually limit the total number of instructions to 2 raised power 6 or 64 but actually a real machine requires more than that. So an extension of this field is actually the last field which is called function field. This part the op part will be same for a group of instructions and it is only the function field which will distinguish them. In fact subtract instruction will also have the same pattern here contrary to what I said. Many of the instructions add, subtract and a few several others will have this part same and it is this part which will distinguish them.

Now, out of these four fields of five bits each (Refer Slide Time: 27:38) three are used for specifying three registers so that is the reason why these are 5-bit field because registers are 5-bit numbers numbering from 0 to 31. In this particular instruction this field is unused what I have written as name for this shift amount which is short for shift amount so there are some shift instructions where there will be something here but in this instruction it is all zeros. Now these three register fields correspond to the three registers in the instruction but not in that order. What appears first in the symbolic form is the destination and that is here rd where rd stands for register destination and rs stands for register source. So s1 is here; this is the code for s1 which you see as 17 (Refer Slide Time: 28:37) and this is the binary code for 17, t0 is numbered 8 so this is the binary code for 8 it is 01000 and rt stands for register third which is this s2 which will have as you can see from this it will have code eighteen which is 10010.

Now this string of 1s and 0s is a 32-bit number which defines this instruction. So add t0 s1 s2 is this number as far as the machine is concerned. Now, rather than writing a long string of 1s and 0s you can also write in compact form in hexadecimal form.

(Refer Slide Time: 29:25)

Machine Language

- lw / sw and the regularity principle?
 - New principle: Good design demands a compromise
- new format (I-type), other format was R
- Example: lw \$t0, 32(\$s2)

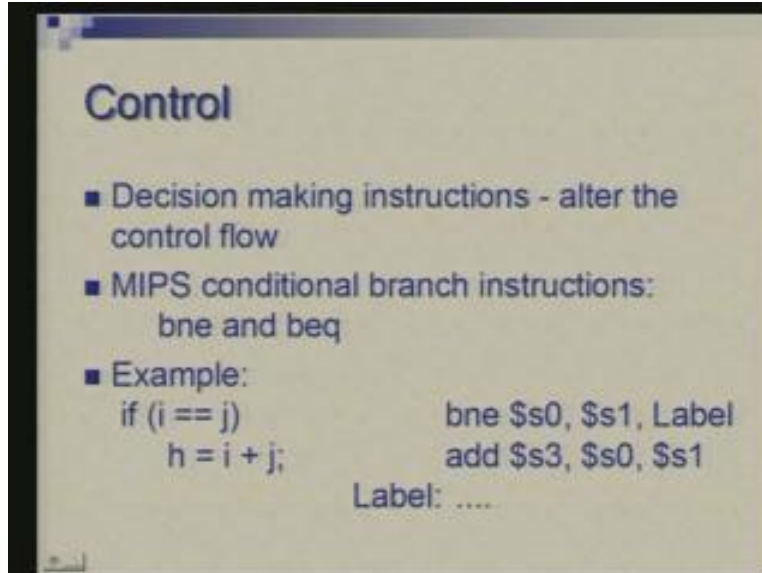
35	18	9	32
op	rs	rt	16 bit number

Going further to load and store instructions we now need to deviate from the format for these instructions. Load store in machine representation we cannot follow same form uniform uniformly and therefore we need to make a deviation from the idea of regularity that all instructions would be of similar nature. So again we need to realize that good design demands a compromise so we cannot be very rigid about this regularity. Here is a new format which is called I format the one which I talked about earlier was called R format R stands for register and I is for immediate; immediate as I will describe later is actually a term used to specify constants.

Let us look at this format now. It has less number of fields basically four fields only. One field is six bits, next two are five bits then there is a 16-bit field. How is an instruction like load t0, 32, dollar s2 expressed in this format. Once again this is an opcode so here we do not have that function field so it is only those six bits which have to be used for defining what the operation is; this is the rs field it corresponds to s2 18 so I am not writing in binary now I will just put decimal equivalents. So you have binary code of 18 laying here in this field corresponding to s2 and well I think this is a this should have been 8 t0 is corresponding to 8 and the offset 32 is put here.

Now basically you would notice here that this constant part which is coming in load instruction has to be a number which is a 16-bit number. It has its own limit. These were arithmetic instructions and instructions for moving data. Now next we go to instruction which define flow of control which allow you to take decisions.

(Refer Slide Time: 31:56)



We have two simple instructions in this called bne and beq. bne stands for branch if not equal and beq stands for branch if equal. To illustrate this let us look at an example of a simple if statement in c. If you are saying if (i compared with j) they are equal then you perform this addition h gets i plus j. So we will make this comparison although it is appearing as equality comparison. What we are actually trying to do is that if they are not equal we are skipping this so that is how we will interpret this in machine language or symbolic assembly language.

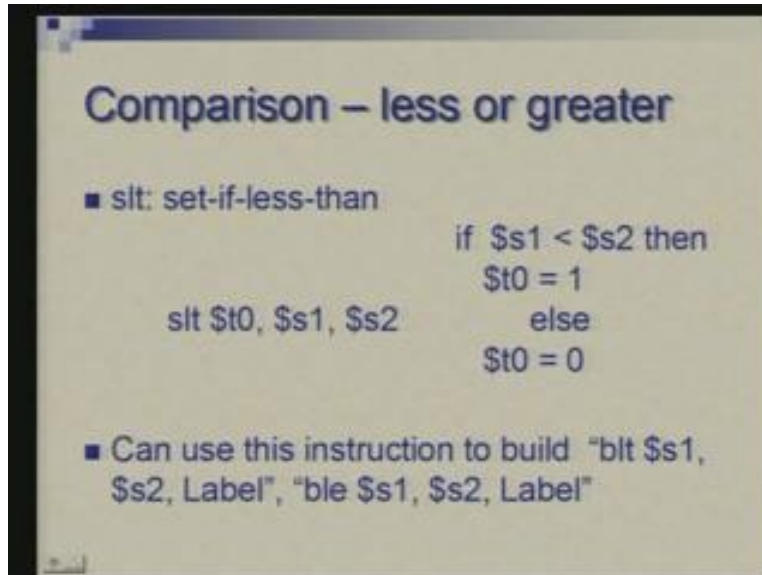
We are saying if s0, s1 are not equal then branch two a statement with this label (Refer Slide Time: 32:53). So statements can have or the instructions can have label which are arbitrary names. this instruction is allowing you to skip the following instruction if the equality does not hold and h equal to i plus j is simply add s3 s0 s1 so i and j are in s0 and s1 and h is going into s3. So whatever instruction follows here will be tagged with this label. This label is linked to that branch statement.

Now, apart from conditional branches which are testing condition we also have unconditional branch which is even simpler; j is the instruction symbol, j stands for jump and label. We are not specifying any operand for comparison here you simply say unconditional jump. We need this in a situation for example if then else. If you are saying i not equal to j you do this else you do that. So here if this condition does not hold we are checking it with beq, if 4 and 5, register s4 and s5 are not equal then we will do this, if they are equal we are branching to label Lab1.

Lab1 is tagged with this instruction (Refer Slide Time: 34:45) subtract which is subtracting s5 from s4 putting the result in s3 and this instruction adds s5 to s4 puts the result in s3. So, after this instruction addition we do not want to do subtract. So, to skip this we need an unconditional jump here, jump to Lab2 the label of the instruction after subtract. After this instruction the control flows either like this to Lab1 and then to Lab2

or goes to add and then jumps to Lab2. So Lab2 is a common point which would be in C language statement. After this here we are either doing this or doing this so this flow of control is organised by one branch which is conditional and one jump which is unconditional.

(Refer Slide Time: 36:13)



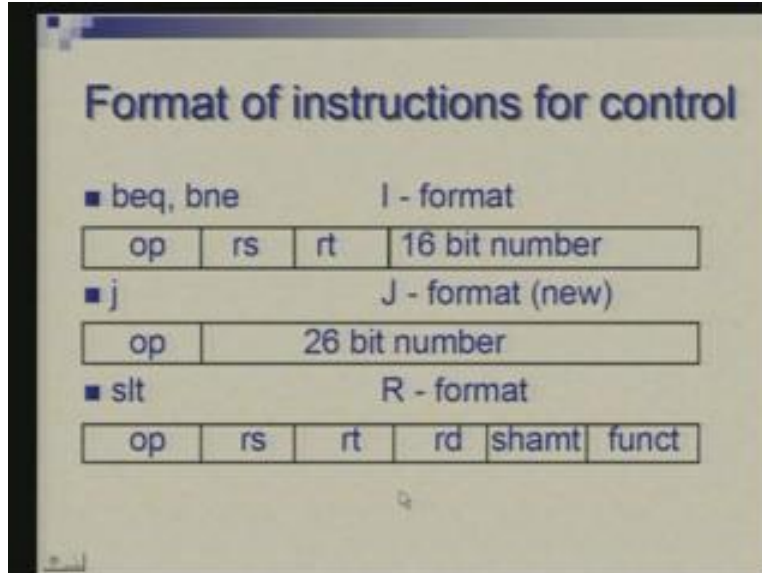
Comparison – less or greater

- **slt: set-if-less-than**
`slt $t0, $s1, $s2`
if \$s1 < \$s2 then
 \$t0 = 1
else
 \$t0 = 0
- Can use this instruction to build "blt \$s1, \$s2, Label", "ble \$s1, \$s2, Label"

Now I talked so far about comparison for equality equality or inequality but you often need to do a less than or greater than type of comparison how do we carry that out? For that there is an instruction called slt which stands for set if less than; slt stands for set if less than. So you set a register to value 1 or 0 depending upon comparison so this instruction for example slt dollar t0, dollar s1, dollar s2 where t0 has a destination where you will set value to 0 or 1 and it compares s1 and s2. If s1 is less than s2 then you set t0 to 1 otherwise you set t0 to 0. Its equivalent definition is this: if s1 is less than s2 then t0 is equal to 1 else t0 equal to 0.

Now this is not a branch instruction; it does comparison but the result is available in a register so it does not alter the flow of control. You need to combine this with beq or bna to achieve an effect of something like let us say blt. Suppose you want to say branch if less than s1 s2 to certain label. In the same spirit as beq we said compare two register for equality and branch if the condition holds. Suppose the comparison is less than and we want to branch you will have to combine a comparison like this and then combine this with a branch instruction or beq or bne types. Basically you will make a comparison, the result is in a register then you check if the register is equal to 0 or 1. We will probably discuss that in a tutorial on how to do that.

(Refer Slide Time: 38:05)

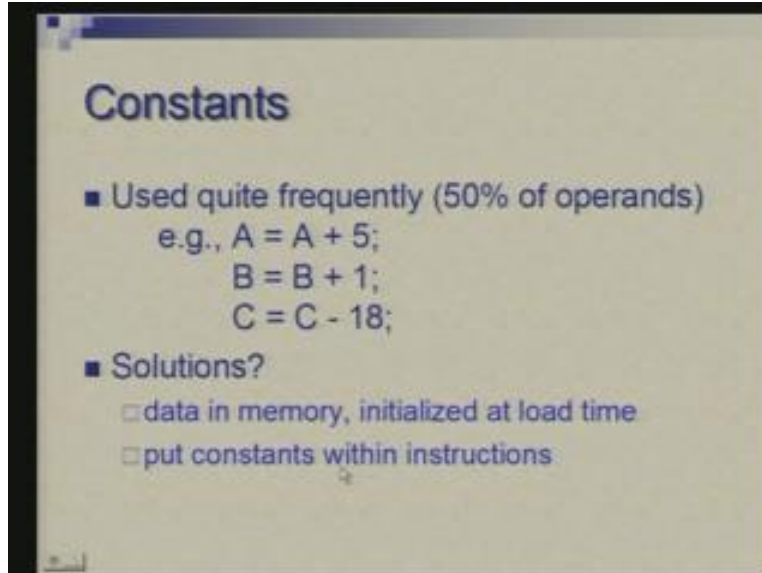


After having introduced this jump instruction let us look at formats of all the instructions we have introduced for decision making and flow of control. `bne` and `beq` follow the I format. There is an opcode two registers which is specified and a 16-bit number which defines the label. The purpose of the 16-bit field is different here. In load store it was an offset in the address from where you have to pick the data; here again in some sense you are trying to define an address but address for instruction.

In assembly form, in symbolic form we have written a label but eventually it actually has to be an address. The number which appears here is essentially an offset of the destination address (Refer Slide Time: 39:03) from the current instruction. I will elaborate on the exact meaning of this later but just remember that this is an offset which you are specifying to define the destination where jump has to be carried out. in the in the jump instruction there is nothing else except for opcode and remaining 26 bits are for specifying the jump address.

`slt` instruction on the other hand goes back to R format where you have opcode, three register fields, shift amount field and function field. Once again shift amount is not being used, `op` and function together define `slt` instruction and there are three register fields.

(Refer Slide Time: 39:50)



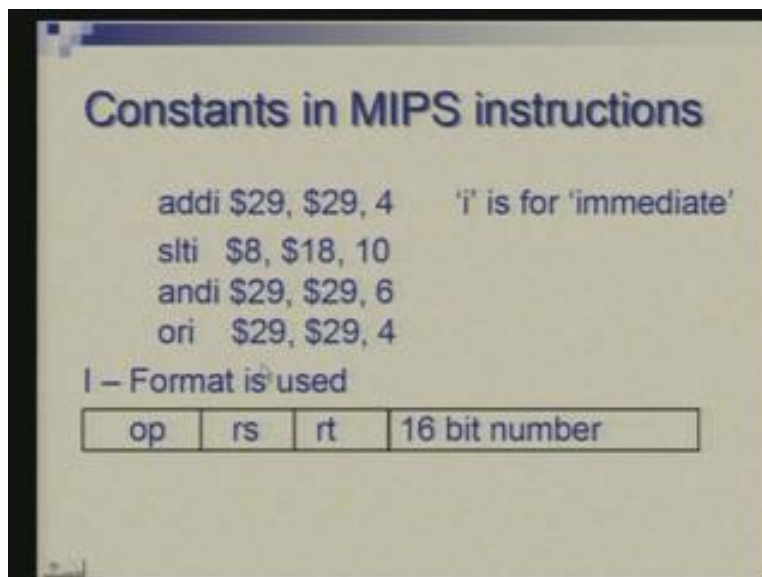
Constants

- Used quite frequently (50% of operands)
e.g., $A = A + 5;$
 $B = B + 1;$
 $C = C - 18;$
- Solutions?
 - data in memory, initialized at load time
 - put constants within instructions

Finally let us move to handling of constants. So you have lot of cases when you have to deal with constants in your computation. So, for example, if you say A gets A plus 5; B gets B plus 1 or C gets C minus 18 how do we do these things? There are various ways it can be done and different processors follow different approach. you can have these constants put as some data in the memory and when a program is loaded in the memory this constant data could also be initialized that could also be loaded.

The other approach is that you can put the constants as part of the instructions. So both possibilities exist in this and we look at the latter one at the moment.

(Refer Slide Time: 40:37)



Constants in MIPS instructions

```
addi $29, $29, 4    'i' is for 'immediate'  
slti $8, $18, 10  
andi $29, $29, 6  
ori  $29, $29, 4
```

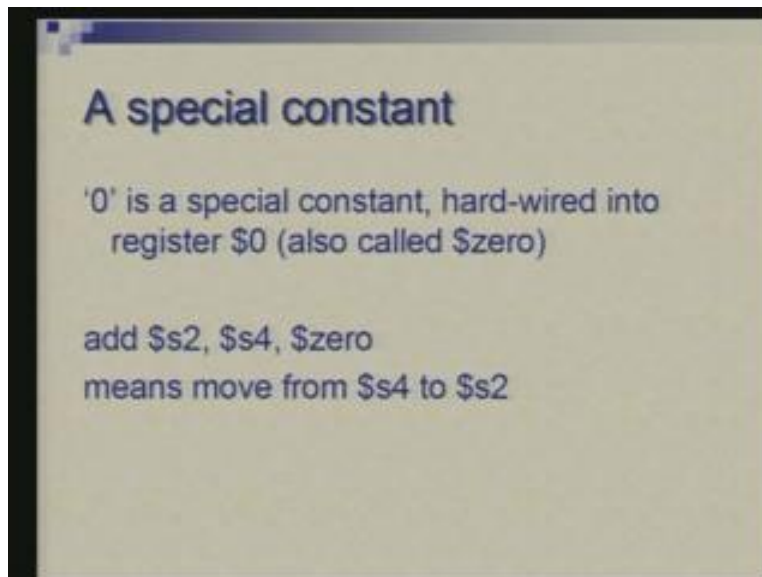
I – Format is used

op	rs	rt	16 bit number
----	----	----	---------------

Here I am putting same instructions but one of the operands and it is the last one is written as a constant instead of a register number and to distinguish it from normal add there is a suffix i which stands for add immediate. Now the reason why we call it immediate is that the operand is immediately available in the instruction itself it does not have to be brought from a register or from memory location so that is the sense in which the term immediate is used. You have add immediate, you have slt immediate and the instruction and and or for logical operation which I have not described the meaning is obvious it is and immediate or immediate and so on and these instructions also follow I format. This is the opcode (Refer Slide Time: 41:33) two registers are being specified and we have a 16-bit for a constant.

Although we deviated from R format to get this constant but you notice that most of these constants are 16-bit numbers. What if you need larger constants we will come to that but there is a special constant 0 which is hardware to register 0.

(Refer Slide Time: 41:57)



So register 0 in among registers 0 to 31 is a special register whose contents remain 0. You cannot alter it by a program. Suppose you write an add instruction with register 0 as the destination so add instruction will just leave it unchanged it will not do, it will perform add operation but the result will be thrown away. So, for example, if you say add if you are using 0 as a source or as an operand if you say add s2 s4 0 it means you are simply moving data from this to this; there is no separate move instruction, add instruction can be equally well used for moving by virtue of this constant 0. It can be written as dollar 0 or dollar zer.

(Refer Slide Time: 42:57)

How about larger constants?

To load a 32 bit constant into a register, we use two instructions

one instruction fills this one instruction fills this

1010101010101010	1111000011110000
------------------	------------------

Now coming back to the large constants which cannot be contained within 16-bits so we should be able to work with 32-bit constants if need be. Now, since instructions deal with 16-bit constant at a time one could have designed for larger design also, one could go for 18, 20, 22 and so on some processors do that but you cannot do you cannot have a single instruction handling 32-bit constant because there will be opcode. You have total 32-bits and you cannot devote all the bits for one operand. So we try to do this with two instructions; one instruction is in the left half left 16 and another instruction fills in the right half or the right 16 bits and we do it by using a special extraction called lui which stands for load upper immediate.

(Refer Slide Time: 43:50)

Loading larger constants

- new "load upper immediate" instruction
lui \$t0, 1010101010101010
- then get the lower order bits right, i.e.,
ori \$t0, \$t0, 1111000011110000

1010101010101010	0000000000000000
0000000000000000	1111000011110000
<hr/>	
1010101010101010	1111000011110000

lui upper means it loads upper part of register left half of a register and i because it is having a constant. So lui dollar t0 and this 16-bit constant this instruction will bring this constant into upper half or the left half of a register t0 in this case. Then we bring in another instruction ori or immediate same register and another constant which corresponds to the lower half. So now this instruction or the contents of register t0 and this constant puts the result in t0. So what is happening is shown here.

After the first instruction is executed register t0 has that constant in the left half, the right half is all zeros. The second instruction is taking this constant and this in the right half left half is 0 the two are odd which means the two are actually put together in this form and the sequence is able to load a constant whose first part is this (Refer Slide Time: 45:15) and the second part is that.

(Refer Slide Time: 45:19)

<u>Instructions</u>	<u>Format</u>
add, sub, addi, subi	R, I
and, or, andi, ori	R, I
slt, slti	R, I
beq, bne	I
j	J
lw, sw	I
lui	I

Now the idea here is that if you have a small constant you can work with single instruction which will be the most common case. But as and when you need larger constants you can spend more time you can use two instructions and do the job. So, to end, let us summarize all the instructions that we have learnt and what is the format used for their binary or machine representation.

We have arithmetic instructions in two flavors with register operand and with constant operands. So one operand of course is a register here also add and subtract add immediate and subtract immediate. Add and subtract follow R format and add immediate subtract immediate follow I format. Then we have logical instruction and and or and their immediate counterpart. Once again the formats are R for and or, I for and immediate or immediate. slt also has an immediate version. the formats are R and I. beq bne are two branch instructions which we have discussed the format is I, j is unconditional branch instruction called jump the format is j so that is the only odd man out here; load word store word also form I format; lui load upper immediate is also I format.

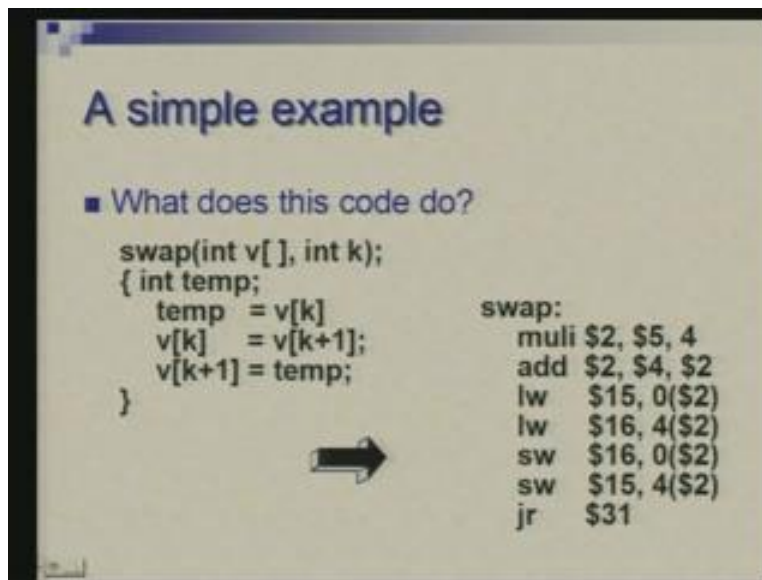
These are the instructions we have learnt. There is little bit you can do with these. You would often need more instructions which we will see in the next class. If you have any questions I can take up now.

Question is what is full form of bne?

bne is branch if not equal, beq is branch if equal. Yes, [Conversation between student and professor..... 47:34] this is from hardware point of view. If you are working with less number of registers, smaller in size, if you are adding smaller numbers the circuitry which can do that is faster; the hardware for adding 232-bit numbers is faster than the hardware for adding 264-bit numbers. Why it is so; we will see later in the course.

Yes please? [Student: if you entered swap once again] Swap instruction, swap operation. Yeah, for swap basically the key part is here.

(Refer Slide Time: 48:22)



These two are load instructions and these two are store instructions (Refer Slide Time: 48:24). So we are reading two words from memory $v[k]$ and $v[k+1]$ into two registers 15 and 16 and writing in the opposite order so they get swapped. What remains is preparing the addresses of memory from where you are going to read. So, before that we actually saw a very simple case where the index of the array is constant. That constant can be placed here (Refer Slide Time: 48:59) in the I format and this goes in that 16-bit field and a register which contains the starting address of A.

A is an array where compiler would put this somewhere in memory. The starting address is we are assuming that before this somewhere we have ensured that the starting address is in s3. Similarly here (Refer Slide Time: 49:22) we are assuming that the starting address of array v is in register 4 and index k is in register 5 so now as far as integers are concerned they occupy one word but our addressing is byte so we want to take starting address of v and add four times k to that. So the first multiply instruction is actually

preparing four times k this contains 5 this contains value of k multiplied by 4 that gets into 2 so register 2 and 4 are added to again bring them to two and that is being used as address here.

Therefore, for v[k] two has complete address you need zero offset, for v [k plus 1] you need additional four offset because v[k] plus means it is 4 bytes ahead after v[k] is it clear any other question?

(Refer Slide Time: 50:35)

Loading larger constants

- new "load upper immediate" instruction
lui \$t0, 1010101010101010
- then get the lower order bits right, i.e.,
ori \$t0, \$t0, 1111000011110000

1010101010101010	0000000000000000
0000000000000000	1111000011110000
<hr/>	
1010101010101010	1111000011110000

So, or instruction (Refer Slide Time: 43:50) basically takes two 32-bit words and performs or operation bit by bit. Here we are using ori that is immediate form of or; it takes one register and one constant. The constant we are specifying is only 16 bits but the actual operation which is performed is a 32-bit operation so the remaining sixteen positions are filled with zeros. So, when the or operation is being done we are taking this number, first number is contained in t0, second number is obtained by prefixing sixteen zeros to this constant which is part of the instruction and the instruction ors is two this is the result which will go to t0 again which is the destination register here. So this is the way we are used to put two parts of a large constant together to form a 32-bit constant in a register.

Any other question? Okay; we stop at this, thank you.