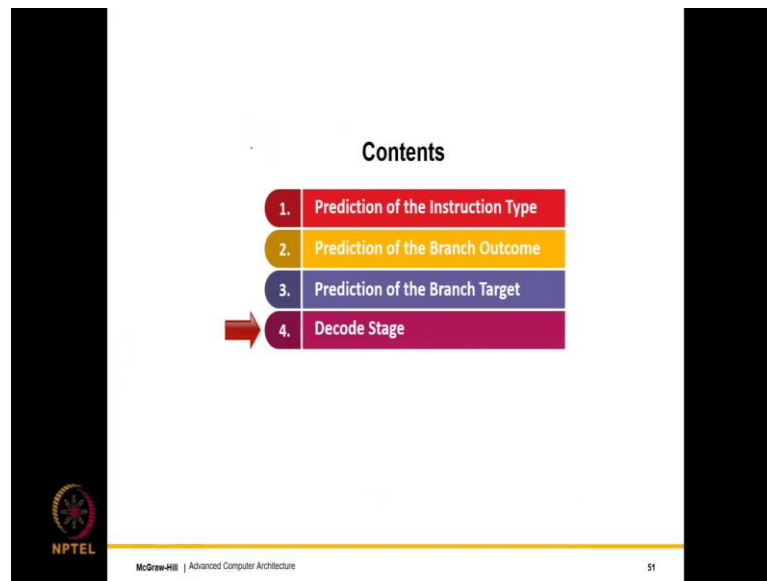


**Advanced Computer Architecture**  
**Prof. Smruti R. Sarangi**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Delhi**

**Module - 03**  
**Lecture - 07**  
**The Fetch and Decode Stages Part-III**

(Refer Slide Time: 00:52)



We will now discuss the Decode Stage which is the last part of the Fetch Unit, and it is also the last part of this lecture, this slide set. So, we will discuss both the decode stage as well as decode time optimizations.

(Refer Slide Time: 01:07)

**The Process of Decoding**

- Expand all the immediate values to 32 or 64-bit values
- Extract all the fields (ids of regs)
- Compute the branch target (if branch is taken)
- Add all implicit sources (such as for the return instruction)
- Create the instruction packet (control bits)

The diagram illustrates the process of decoding an instruction. It shows a 32-bit instruction packet with fields for 'imm' (immediate) and 'offset'. The 'offset' field is used to calculate the branch target as  $PC + \text{offset}$ , which is then expanded to a 64-bit value. The 'imm' field is also expanded to a 64-bit value. The diagram also shows the 'ra' (return address) field and the 'control bits' field.

NPTEL

McGraw-Hill | Advanced Computer Architecture

52

So, decoding typically in most processes involves the following 5 activities. So, in most instruction sets, the instruction contains an immediate field inside it. So, if it is a RISC instruction set, the size of the immediate field is limited. So, for example, let us say it is a 4 byte instruction set, so there are 32-bits.

So, in this case the immediate field clearly cannot take up all of 32-bits, so maybe it can be a maximum of 16, 17, 18-bits. So, in the simple RISC instruction set the immediate field is 16-bits and there are 2-bits for modifiers, right. So, that is the format that we have. But in different instruction sets the immediate field will have different sizes.

So, regardless of the size in the decode process, the immediate field is expanded to whatever is the width of the data path of the pipeline. So, it is a 32-bit machine which means that everything inside the pipeline is 32-bits this includes registers, it includes the values that are read and processed and processed by functional units and so on.

So, the immediate field is sign expanded to make a 32-bit field. Similarly, if it is a 64-bit processor, then the immediate field in this case will be expanded to a 64-bit field inside the processor.

The rest of the fields which are the ids of the source and destination registers all of these fields are extracted, so we can see these are the ids of registers can be source, can be destination. Furthermore, the branch target is also stored within the instruction if it is a

branch. So, most of the time for the branch targets, we are actually looking at what are called PC relative branches.

So, a PC relative branch the target is essentially the program counter plus an offset and the offset is embedded within the instruction. So, if this is an instruction over here, then a part of it will record the offset. And so, when this is added to the program counter, so that is, so that are in program counters, the PC plus offset becomes equal to the branch target. So, in the decode field typically, if we are using an offset based branch target, the target is computed.

Additionally, we also process special instructions, for example, the return instruction is a special instruction which has an implicit source. Implicit source means that the source is not named, but still there is a source register. In this case, it is the ra register, it is the return address register. So, this is at this point explicitly added to the instruction packet, and with all of these fields and everything all expanded the instruction packet is created.

And most of the time if you are creating a hardware implementation, we also add many more pieces of information, these are typically called control bits, which control the processing of the instruction in later stages. So, a later stages let us say which functional unit to choose, whether it is a memory instruction or not. So, a lot of this information is at this point of time also added to the instruction packet, such that subsequent stages will find it much easier to process the current instruction.

(Refer Slide Time: 04:56)

### Issues with CISC Instructions



**CISC Instructions**

- Do not have a fixed length
- In **x86** the length varies from 1 to 15 bytes
- It is **hard** to fetch multiple instructions at once
- Need to know the boundaries of instructions
- It is hard for OOO pipelines to process them
  - Most CISC processors internally convert from CISC to RISC

*RISC → same size*

*0, 4, 8, 12*

```
graph LR; CISC[CISC instructions] --> Decode[Decode unit]; Decode --> RISC[RISC instructions μOPs]
```



McGraw-Hill | Advanced Computer Architecture

53

So, now let us come to CISC instructions, Complex Instruction Set Instructions. So, CISC instructions, but they create a lot of problem and the stage in which they create the maximum amount of problem you can see the scorpion here, it is actually the decode stage. So, the primary problem here is that the fetch stage basically reads in a set of bytes and in the decode stage we actually need to break down the instruction, under break, understand and break down. The sad part is that instructions do not have a fixed length say.

Let us say if you were to read 32 bytes from the instruction cache, in the case of a RISC instruction set it is very easy. So, we know that every instruction is of the same size, right in a RISC ISC. We know that all instructions are of the same size 4 bytes or 8 bytes.

And we also have a strict alignment, in the sense, if instructions are 4 bytes long then they started memory addresses that are multiple of 4 bytes, like 0 bytes, 4, 4, 8, 12 and so on. Say, clearly know where an instruction starts and how long it is. So, this is why RISC instruction sets, processing them is much easier and their decode units are also very simpler.

In comparison, let me just erase ink, yeah. In comparison for CISC instruction sets, like the Intel and AMD x86, the length of an instruction varies from 1 to 15 bytes, right. So, particularly in Intel x86, what we can find is that we can have very short instructions like an OP instruction for example, and we can also have very long instructions that have complex addressing modes and which actually achieve a lot, and these will be extremely long.

(Refer Slide Time: 07:10)



**Issues with CISC Instructions**

**CISC Instructions**

- Do not have a **fixed** length
- In **x86** the length varies from 1 to **15** bytes
- It is **hard** to fetch multiple instructions at once
- Need to know the **boundaries of instructions**
- It is hard for OOO pipelines to **process** them
- Most CISC processors internally convert from CISC to RISC

*\* decoding is sequential*

*fetch*  
CISC instructions **outside** → Decode unit → RISC instructions (μOPs) **inside**



McGraw-Hill | Advanced Computer Architecture

53

So, the instruction size can be long and variable, it is clearly not fixed. Hence, given a set of bytes finding the boundaries of instructions, right, you know they are of different sizes finding out where they are, is actually quite tough. So, what we need to do is that let us assume that the first instruction starts at this point, we need to keep on reading the bytes, keep on decoding and find the end of the instruction, again start reading the bytes.

So, this constraints our bandwidth because decoding becomes a sequential process. We need to start from one end, find the boundary of the first instruction, then find the boundary of the second, then the third, and then the fourth. So, unlike RISC instruction sets, where we can read let us say 16 bytes and if an instruction is 4 bytes we can decode in parallel. In the CISC instruction set, decoding is fundamentally a sequential process and this is what causes the issue.

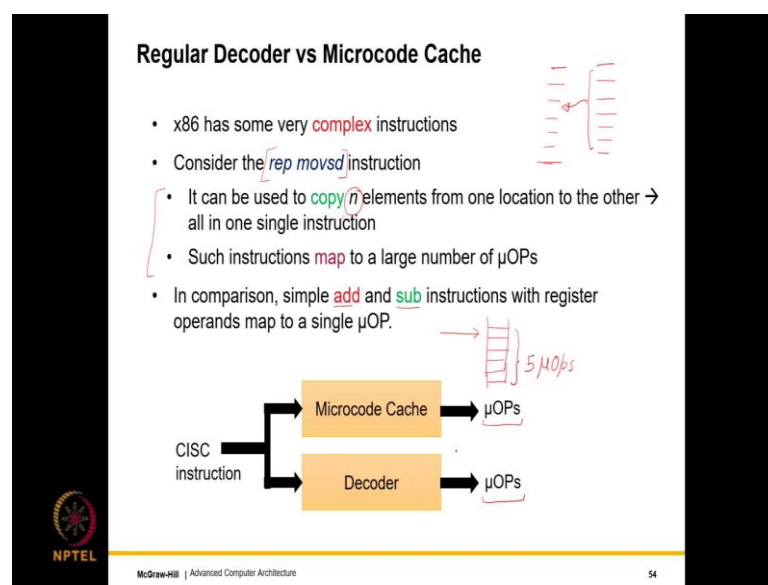
So, what is the issue? Well, the issue is that decoding in a CISC ISC, unless we do something extra, decoding is a sequential process which is not acceptable to us. Because this means that we cannot process multiple instructions in parallel, right. And the main reason is we do not know the boundaries between the instructions.

Furthermore, if instructions are that complicated that one instruction does not do anything, and one instruction does a lot, it is very hard for out of order pipelines that actually rely on the regularity and uniformity of instructions to process such complex instructions. So, that is why in most CISC processors, notably Intel and AMD, the fetch

unit does read CISC instructions from the instruction cache, but internally the decode unit converts the CISC instructions to internal RISC instructions.

These internal RISC instructions are known as micro OPs or microcode. And this process of conversion is done unbeknownst to the programmer. But this does happen, it has to happen, such that we can gain extract the maximum amount of performance from the out of order pipeline, right. So, it is essentially CISC outside, right. So, essentially Intel or AMD processor is CISC outside, RISC inside.

(Refer Slide Time: 09:44)



So, let us come to x86 and show the example of one of the instructions which can actually become very complicated. It is the rep movsd instruction. So, what it does is that, it embeds an entire loop within one instruction itself. So, it does several things and what it does is that it can be used to copy n elements from one location to the other. So, the locations are implicit, right, two registers ESI and EDI actually.

And from here what can happen is that, if let us say from one memory location we can take a bunch of data and how many elements will n itself is stored in another register which is also implicit. So, in this case, all of this data is moved to another memory location. So, it just copies one array of bytes to another array and this is achieved by a single instruction.

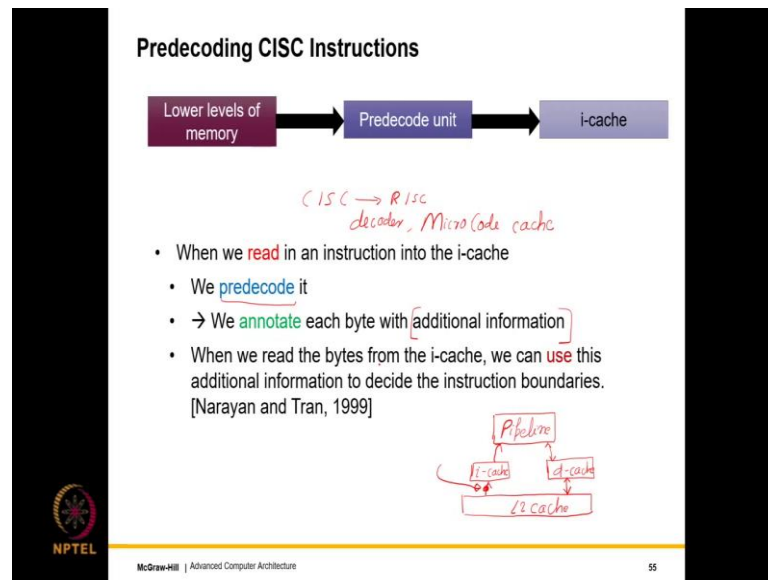
And so, when we actually convert from CISC to RISC, this instruction will map to a very large number of micro operations, ok, large number of micro operations, primarily because an entire loop we are compressing that into a single instruction. In comparison in a CISC ISCs, simple add and subtract instructions with register operands will map to a single micro operation.

So, to summarize our discussion, once we read in a CISC instruction, once we fetch a CISC instruction, if let us say it is a simple; so, we can now divide a complex instruction set to a simpler complex and a more complex. So, if it is reasonably simple, like add, subtract, multiply, divide, etcetera, it is sent to the regular decoder, and the regular decoder provides a stream of micro operations.

And if it is complex, very complex, like they have move movsd, so this is not the only complex instruction that does a lot there are many more such instructions in x86, and for all of them we actually have a table. So, this table is known as a microcode cache or a microcode table. So, in this case, what happens is we addressed the table by the instruction and it is a huge table. So, at this point we just read all the micro operations that this instruction would translate to.

So, in this case, if it let us say translating to 5 micro operations, what we do is that we simply read them and these micro OPs are sent down the pipeline. So, the microcode cache for every complex instruction stores a set of basic micro instructions and it is necessary to read this cache for effective translation. So, we do not use the decoder for complex, for extremely complex instructions we just use the cache otherwise we use the decoder. So, keeping this in mind, let us move forward.

(Refer Slide Time: 13:09)



So, we need to understand that even though we have solved one problem which was the fact that out of order pipelines would find it very hard to process this instructions. So, we converted CISC to RISC. So, this is what we have achieved up till now. And how did we convert CISC to RISC?

Well, we have a decoder which does that. And for some very complex instructions we also have a microcode cache where we can just read the micro instructions for a given CISC instruction. But our original problem of finding out the boundaries of instructions in just a raw stream of bytes that we read from the instruction cache, this problem has not been solved, right. So, to solve this we will refer to a patent Narayan and Tran.

So, this is a very famous patent and different versions of this patent. So, you will find different versions, extensions, or alterations, augmentations of this idea in various forms, but pretty much modern processors do something of this nature to handles CISC constructions, otherwise there is no other way. So, what we do is that to find the boundaries whenever we read in an instruction into the i-cache.

So, here it is important for us to talk a little bit about the memory hierarchy in typical processing systems. If you have the pipeline over here we have separate hardware structures, one of them is the instruction cache which we have also referred to as instruction memory in the past and we also have a data cache. So, mind you these are separate physically separate structures.




So, from the instruction cache only read, data cache we read and write. So, they of course, contain a subset of the memory locations. Then we have an L 2 cache. So, L 2 cache is a level 2 cache which contains both instructions as well as data, right. So, and below that we can have additional levels as well. So, it does not matter. What we care about is we care about this step where we are reading an instructions into the instruction cache, right.

So, when we are doing it what we do is we have a little decoder over here which we call a pre-decoder, alright. So, we have a little decoder sitting over here which we call the pre-decoder. Here, if let us say we read in 16 bytes or 32 bytes at a time, we annotate each byte which means the instruction cache has some additional bits for each byte, right, for each instruction byte we have a little bit of extra bits which store some amount of additional information.

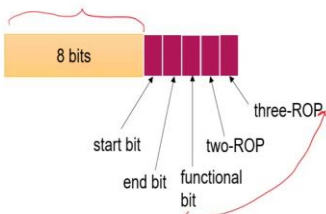
So, this additional information can be used to decide many things, one among them is finding the boundaries between instructions, right; where one instruction ends and where one begins. And this of course, is not a problem in RISC, but this is a massive problem in CISC, alright. So, let us see.

(Refer Slide Time: 16:33)



NPTEL

### Predecoding - II



- Start bit → Starting byte of an instruction
- End bit → Last byte of an instruction
- Functional bit → Interpretation depends on the implementation.
- two-ROP and three-ROP bits → Number of  $\mu$ OPs in an instruction

*1) CISC  $\rightarrow$  RISC*  
*ii) Predecoding*

McGraw-Hill | Advanced Computer Architecture

56

So, this particular patent when Narayan and Tran in 1999, what this says is that for every instruction byte which is the 8-bits, we create we use 5 more bits of information which if you think about it is a sizable overhead, but this is actually required for performance

reasons. So, the 5 bits are as follows, one is the start bit, this indicates if this is the starting byte of an instruction. So, clearly this is indicating a boundary.

Also, we have an end bit, which indicates if this is the last byte of an instruction. So, between the starting between the start bit and the end bit, lie all the instruction bytes. So, this can be very efficiently retrieved, right. So, you can find out boundaries very easily. Then we have a functional bit, so the patent says you are you are welcome to read the patent.

So, the patent says that the interpretation depends on the implementation. So, one implementation could be, right, one of the implementations could be that this specific instruction if this bit is 0, it means that the decoder will be used; if it is 1 it could mean that the microcode cache will be used, right. So, this can be one use of this bit.

Then we have two more bits called two-ROP and three-ROP bits. They essentially refer to indicate the number of micro OPs that this instruction will translate to, alright. So,, but of course, in any processor this format would change in the sense we might; so, let us say particularly the last 3-bits, right; this format will change. We might use want to use these bits for other purposes.

So, this is very implementation dependent. But at least the first 2-bits, clearly help us identify a boundary. See, if we have start bits technically speaking an end bit is not required. However, it does make processing faster and at the hardware level every cycle saved is significant.


(Refer Slide Time: 18:55)

### Optimizing Operations on the Stack Pointer

**Example code**

```
sub sp, sp, 8  
st r1, 0[sp]  
st r2, 4[sp]  
....  
ld r2, 4[sp]  
ld r1, 0[sp]  
add sp, sp, 8
```

- There is a **pattern** here.
- We either **modify** the stack pointer by adding or subtracting a constant
- The **load** and **store** instructions access the stack pointer.
- Update the stack pointer locally or get the memory address directly



Have a copy of the *sp* register in the decode stage itself.

NPTEL

McGraw-Hill | Advanced Computer Architecture

57

Now, let us look at the second decode time optimizations. We already have looked at one which is pre-decoding. So, what does pre-decoding give us? Well, let us take a quick detour to the previous slide. So, what CISC to RISC conversions? We looked at two ideas basically. The first idea was CISC to RISC conversion. So, what this gives us is it gives us a nice homogeneous uniform processing environment within the pipeline and this is something that we needed, alright.

So, because as you will see when you see the design as out of order pipeline that a lot of homogeneity is required, is desirable, and is preferred, because this allows us to process instructions quickly. The other was idea of pre-decoding which all also absolutely essential, it helps us find the boundaries between instructions.

How do we do it? Well, when we are populating the i-cache by reading in data from lower levels of memory, at that time we pre-decode the instruction which is a much it will be a much smaller unit with the regular decode unit which will just traverse the bytes and figure out the boundaries.

Now, let us look at another kind of optimization called optimizing operations on the stack pointer. So, look at this piece of code. This piece of code is representative of what we would write in a function, normally. So, what we are doing is that we are creating space in the activation block. So, we are subtracting the stack pointer by 8, so this is creating 2 slots for us.

So, then we are doing registers spilling source, registers r 1 and r 2 they are being spilled into the memory location. Then, we might be doing 100s of things calling functions, those functions might be calling other functions, it does not matter. We are then restoring the registers, and finally, in this line we are deleting the stack frame, alright. So, they are deleting the activation block. So, whatever space was created that is being deleted.

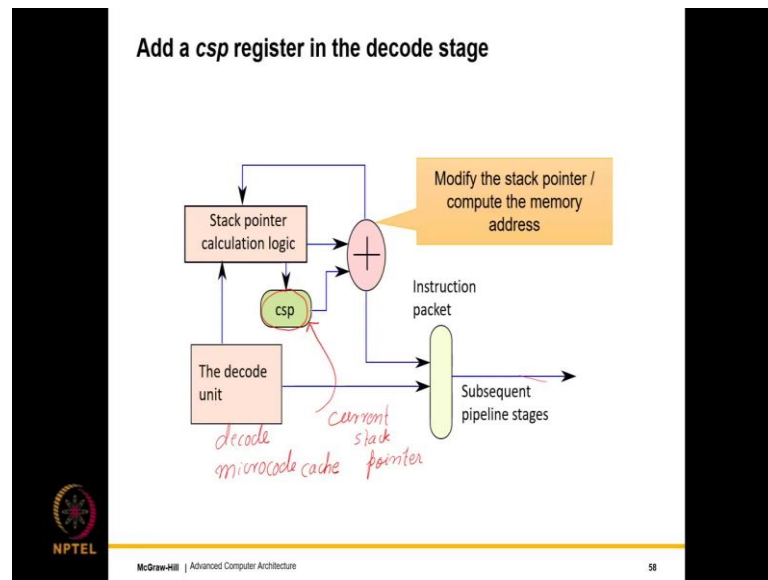
So, this if you see is a rather common pattern. But another pattern if you see is that we either modify the stack pointer by adding or subtracting a constant, right. This is the first pattern. So, if we look at all of them, we are essentially reading in the contents of the stack pointer adding 0 in this case, adding 4 in this case, adding 8 in this case. So, in all cases we read the contents of the stack pointer and do some add subtract.

The other is that the load and store instructions access the stack pointer. Of course, and then add and subtract small constants, compute a new address and then go and access that address.

So, why not? Here is the idea that update the stack; so, create a; so, since we any way fetch and decode instructions in program order, why cannot we create or keep a small copy of the stack pointer in the decode unit itself. And anytime we find an instruction of this nature we quickly subtract 8 from the stack pointer, anytime we find an instruction of this nature we quickly add 8 to the stack pointer, right.

So, what we do is instead of treating the stack pointer as a regular register we keep a small copy of it in the decoder. And most of the instructions on the stack pointer as you can see there are many of them, they can quickly be done at the decode stage. And since we are only talking about one register not a register file, these accesses will be fast. So, how will the scheme look like? Here it is.

(Refer Slide Time: 23:03)



We add a current stack pointer register in the decode stage. So, this is the crux of the idea. So, we have a regular decode unit that contains the decoder and the microcode cache. So, this of course remains the same, no changes to that. So, we have a very quick mechanism of detecting, if the stack pointer is being accessed. So, once we can very quickly see that this is happening, the instruction is additionally sent to the stack pointer calculation logic in the decode stage of course.

And if we take a look at this piece of code, we will see that in almost all cases we read the value of the stack pointer and either add or subtract some constant to it. This value is applied to the rest of the pipeline.

So, in this case, this is actually a memory address. And so, this memory address needs to be sent to the memory unit, in this case the data cache for reading the value. So, regardless of what we want to use the data for, the address can be sent down the pipeline. But let us say in this instruction we add 8 to the stack pointer and then we use it to update the value. So, this can also be done.

So, what we do is that we quickly extract the immediate field, and we also read the value of the current stack pointer, we add both of them. After adding that if the stack pointer is not being updated, well then, the result of this addition is added to the instruction packet which is subsequently sent to the rest of the pipeline stages. Otherwise, if this is not the case then the stack pointer, if let us say if that is being modified, well then, what we do is

that we send it back to the stack pointer calculation logic and this piece of circuitry overwrites the value of csp.

So, csp is essentially the central state element over here that stores the value of the current stack pointer. And whenever it is accessed or modified we can use this quick logic to take care of it. And so, why is this an efficient idea? Well, it is efficient because most of the time in a very large number of instructions, we actually access the stack pointer

So, as you can see over here for a large number of load instruction, store instructions are creating and destroying the stack frame, we use the stack pointer and pretty much the format of instructions that use the stack pointer is the same, right. So, they are used roughly in the same way. So, that is the reason it is good to have a csp register right here in the decode stage such that the rest of the pipeline does not have to bother with processing these instructions.

(Refer Slide Time: 26:12)

**Corner Cases**

`ld sp, 12(r1)`

- When we encounter such instructions, we set the value of `csp` to null.
- Treat `sp` as a regular register in the OOO pipeline.
- Accumulate the difference,  $\Delta$ , between the value returned by the load `addr` and the current value of the stack pointer.
- When the load instruction returns with its value: set

$$csp \leftarrow addr + \Delta$$

processor  $\leftrightarrow$  compiler

$\Delta = 16$

`ld sp, 12(r1)`  
 $\leftarrow sp = addr + \Delta$   
`add sp, sp, 8`  
`add sp, sp, 8`

NPTEL

McGraw-Hill | Advanced Computer Architecture

59

So, what are the corner cases? Well, we could have an instruction of this type which does not fall within our format. So, in this case, we read some other value and we store it in the stack pointer. So, since this does not fall within our format means, but we still need to handle it because it is a valid instruction.

What we do is when we see such an instruction we set the value of csp to null, and then we treat the stack pointer as a regular register, make the instruction pass through that out of order pipeline, and then at some point of time the out of order pipeline will set the value of the stack pointer, right.

Till that point what do we do? Well, till that point we will have instructions of this type that update the stack pointer. So, till that point what we do is we just accumulate the difference, right. So, assuming the value that is read 12 r 1, assuming the value that is, so a 12 r 1 in this case, since you are writing to the stack pointer and the stack pointer points to an address, so we are essentially reading an address from here.

So, let us say that the load instruction here will return a value which we call addr. Say addr primarily signifies that what we are reading from 12 r 1 is actually a memory address, the stack pointers are supposed to store memory addresses. So, let us name our temporary variable as addr. So, what we do is that we accumulate the difference delta between the value returned by the load and the current value. So, let us say, let me explain with an example.

Let us say, we have this instruction over here, and here the value of the stack pointer, right, whatever is returned is essentially is sp is equal to this value, right. So, then we could have instructions of this form. So, let us say at this point, the value of the stack pointer comes back. So, what we do is that delta will be equal to 16, the reason being that we add 8. And then of course, we do other things we do not touch the stack pointer again we add it, so delta will be equal to 16.

So, we set the value of the current stack pointer which is csp equal to the value returned by this load instruction which is addr which we are assuming it is, right. This is just the name that we are giving. It has no architectural significance; plus the accumulated difference which is delta in this case. For this example delta is 16.

So, we can add both of them, we can update the csp, and then our regular process can continue. So, the advantage over here is that we are sending as few instructions involving the stack pointer to the out of order pipeline as possible. And nevertheless, we will have corner cases that is why they need to be handled.

But a smart compiler will not generate code like this because it would know that this would involve, this would entail additional work on the side of the processor, and hence it is not desirable to produce this kinds of code, so it will not do it. So, this further tells us that there is a rather intimate connection between the processor and a compiler.

So, the compiler should know which instructions the processor executes well, executes quickly, and which instructions cause a problem, right, a performance problem. So, this clearly causes a performance problem because this is an instruction that is sent down the pipeline and the stack pointer registers which we normally, so sort of take care in the decode stage we will not do that. So, essentially a full instruction needs to be processed.


So, in this case a good compiler will actually not generate such instructions it will instead generate instructions of this type, such that we can take care of these instructions, handle these instructions in the decode stage, alright. So, one point is drew here that let us say we have an instruction `sub sp, sp, 8`. So, in this case we can directly add 8 to csp and update its value in the decode stage. Subsequently, this instruction can be discarded or this instruction can be dropped, because its job has already been done.

So, since it has been fully executed, in this sense it is updated the csp register, there is no need for it to occupy additional resources within the pipeline and it can be discarded right here itself. So, this is one of the many advantages of this scheme. And similarly, here also we are not using the heavy duty adders in the pipeline, we are just doing a quick addition over here.

So, this is saving us a lot of effort in the pipeline. And furthermore, if let us say an instruction of this kind would go into the pipeline what we will see is that it will involve a lot of processing, but in this case we can sort of, it is like a shortcut. So, we can do things rather quickly.



(Refer Slide Time: 32:07)



NPTEL

### Advantages

*ld r1, 8[sp]*  
*sp+8*

- Load instructions to the stack can be issued **early** because we compute the address at decode time.
- For **load** and **store** instructions to the stack, the address need not be computed in the pipeline.
- We can get **rid** of such instructions. This will effectively reduce the dynamic instruction count in the rest of the pipeline.

*[add sp, sp, 8]*

McGraw-Hill | Advanced Computer Architecture

60


So, given this what are the advantages? Well, advantages are many, we did discuss a few a few more. So, consider a load instruction, right. So, in this case, let us say you are loading it from this address. So, what we are doing in the decode stage is that we are computing this address which is essentially  $sp + 8$ , right and  $sp$  stored in  $csp$ .

So, it is  $csp + 8$ . So, since we are computing this address, we do not have; so, we are essentially saving several operations in the pipeline. So, in the in the rest of the pipeline. So, we are not reading the value of the stack pointer and we are not performing the addition.

We have an address, and this address can be sent directly to the memory. So, we will see there are other conditions later on. But these load instructions to the stack can essentially be issued early because we already know the address, and as we will see if some other conditions hold which we will see in a later which we will see in chapter 4, but if those conditions hold then this address can be sent to the memory system early and the read can be performed.

Similarly, for load and store instructions, there is no need to compute the address in the pipeline, they are being computed in the decode stage itself. Finally, we can get rid of some of these instructions. For example, add so, we can essentially get rid of it. What this would mean is that since this job is done, there is no need for additional processing in the pipeline.

(Refer Slide Time: 33:56)



NPTEL

### Instruction Compression

- Instruction caches have a **limited size**: typically 32 to 64 KB
- The performance is extremely sensitive to their size
- Hence, we wish to **pack** as many instructions as possible
- **Approach 1: Reduced-width instructions** (*Thumb*)
  - Support a limited number of opcodes
  - **Avoid** encoding complicated flags and options
  - Reduced view of architectural registers *16 → 4    8 → 3*
  - Reduce the size of the immediate fields
  - Implicit operand (accumulators):  
 $\text{add}(r1, r2)$                        $(r1 \leftarrow r1 + r2)$   
*add r1, r1, r2*  
*[add r1, r2]*

McGraw-Hill | Advanced Computer Architecture 61

Now, let us look at the third optimization called instruction compression. So, in instruction compression what happens is, so we will have to understand that instruction caches have a limited size. So, they are between 32 to 64 kilobytes typically. And the performance is extremely sensitive to their size, particularly, for cloud computing workloads where they anyway have a lot of instructions.

And so, a little bit a small change in the efficiency of the instruction cache can actually cause the create a major difference. Hence, we wish to pack store as many instructions in the instruction cache as possible.

So, what are some of the approaches? So, one approach is that we have a reduced width instruction which in any case the r1 ISC does support. So, r has the thumb instruction set, the thumb is so instead of an r, it is like a thumb which means it is a much smaller instruction set. So, this allows us to essentially compress it, reduce the number of bits basically, and store 30 or 40 percent more instructions, for an instruction in an instruction cache with the same size.

So, how is this compression achieved? Well, the way that we achieve this compression is that instead of supporting all the opcodes, we support a limited number of opcodes, the ones that are the most common. For example, add, subtract, load, store, call, return, etcetera. So, we just support a very small subset of it in a reduced instruction set. We do not encode complicated flags and options. Also, in the regular r1 ISC we can see 16

registers with the thumb ISA we can see only 8. So, we see a reduced view of architectural registers. So, this basically reduces the number of bits that are required to encode the registers.

For example, if we are seeing 16 registers and for each register we need a source or destination we need 4 bits. But if you are only looking at 8 registers, then we actually need 3-bits. In many cases, we do not need that many registers. So, we can afford to live with a reduced view of the number of registers.

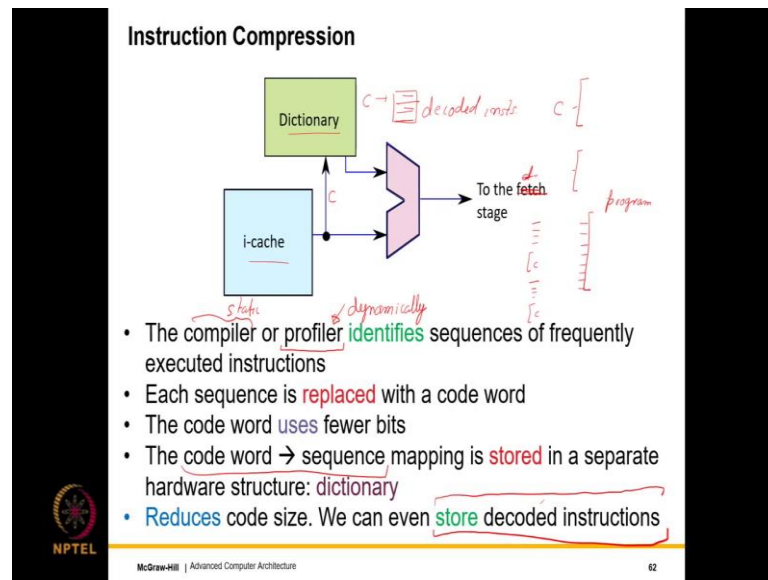
Similarly, we can also reduce the size of the immediate fields. So, there is no need to embed very large constants in the instruction. So, this also can reduce the size. Furthermore, we can have instead of having 3 arguments in the instruction, we can have 2. So, such a style is also known as, you know this approach is called, an accumulator based approach for register  $r_1$  is called the accumulator because the operation that is actually performed is  $r_1$  is equal to  $r_1$  plus  $r_2$ .

So,  $r_1$  is both the source, is both a source and a destination, and the destination is of course, implicit. But what is happening is  $r_1$  is equal to  $r_1$  plus  $r_2$ . So, ideally we would have written  $r_1, r_1, r_2$ , but since this is a reduced instruction set we can write  $\text{add } r_1 \text{ comma } r_2$ , alright.

So, these are several methods by which we can take a regular RISC instruction set or a regular CISC instruction set and create a reduced instruction set, such that to store an instruction it takes requires less bits, less bytes. This would further translate to an increased instruction packing density in i-cache which will give us tremendous benefits.

So, much of this we will understand when we go to chapter 7, when we actually see, when we actually see how caches are designed and how the memory system works. But till now you would simply need to believe me that if you can store more instructions in the i-cache, it is always better. And one way of doing this is this where you can clearly see that this is a fairly effective method of increasing the storage density within the i-cache.

(Refer Slide Time: 38:37)



One more idea. Well, one more idea is that typically we have large sequences of code that are similar across the program. For example, code to read and write registers that is similar. To read and to call certain known functions, to perform some operations like, so in so in every program will have some common operations that are actually performed at different places.

So, the compiler and what is called a profiler, so a profiler is a special piece of software that pretty much runs a program once with representative inputs. It finds, then the profiler takes a trace of all the instructions that are executed.

So, there are different ways of doing that, but this is too premature a chapter to discuss how to do it. But let us assume that the provider has some way of seeing all the instructions that the program is executing, then it can identify sequences of frequently executed instructions, right. So, the compiler can do it at compile time which is statically and the profiler can do it at runtime which is dynamically, alright.

So, regardless of how it is done, ultimately what will happen is that be it static or be it dynamic, what will happen is that we will find within the program, we will find chunks of instructions that are actually the same, right. So, one obvious source is when we are reading or writing registers, spilling registers basically while entering a function, right. So, that is one of the places where definitely the code will be the same. And it will also

be same in other places where maybe very similar arithmetic operation or a similar logical operation is being performed.

What we do is that for all of these instructions we replace them with a code word. So, the code word is like a very small mnemonic for the entire sequence. And the code word will be stored somewhere, but at least it can be used to compress the instruction sequence. So, we will have regular instructions, then we will have a code word, again regular instructions, then a code word.

And the processor can use some bits of the code word to actually identify that it is a code word, but one clear advantage is that we will be using fewer bits and this will also increase the instruction storage density in the i-cache.

So, how does the processor handle the code words? Well, internally it of course, has the i-cache. And it has a dictionary which is something similar to the microcode cache. So, in this case whenever a code word is encountered, and so, whenever we read a code word from the instruction cache, the code word is sent to the dictionary.

The dictionary looks up the code word and outcome a sequence of instructions, these sequence of instructions are then sent to the sorry, this should be the decode stage, right. The, I am sorry this is actually the fetch stage because there are other things that the fetch stage also does, so this is fine. So, this is sent to the rest of the fetch stage, ok. So, to the rest of the fetch state this instruction is sent.

So, the code word as I said is a sequence, the code word to the instruction sequence mapping is stored in a separate hardware structure which we call the dictionary. And the advantage here is, so there are two advantages one is that using intelligent compiling and profiling, we find a way of reducing the both the length of your program and the instruction cache density, right and the instruction density within the i-cache. This is one.

And the other is, so this of course, reduces code size, both static code size and dynamic code size and along with that what we can do is instead of storing regular instructions, we can store decoded instructions, right. So, then the advantage is that these instructions do not have to go through the decoder, they can directly go to the stage after decode which is renaming because these instructions are already decoded, so there is, so this saves time as well as it saves power.

So, we will see that Intel did extend this work. And so, of course, it did not do compression, they did something else, it created something called a trace cache which stores long sequences of decoded instructions to effectively bypass the fetch and decode stages. So, we look at a trace cache later on in chapter 7, but this is just a quick trailer for the movie.

So, what again is the basic idea. Well, the basic idea here is that instruction sequences are replaced with code words, and code words are naturally much shorter. So, we clearly see a code size reduction. And the code size reduction, well if the compiler is doing it we will see both static and dynamic time reduction. And also, if the profiler is doing it will see a reduction, but the profiler of course has more information because it is taking a look at the dynamic sequence of instructions as opposed to the compiler.

Regardless of who is doing it, we can increase the efficiency of this process by also storing instead of storing regular instructions we store decoded instructions. So, essentially we store the micro OPs. So, in this case, we can bypass the decoder and get a performance advantage from there, alright.

So this completes the fetch the chapter on fetch and decode stages. So the next chapter which is chapter 4 we will focus on the issue execute and commit stages of an out of order pipeline.