9 Designing an Instruction Set

9.1 ANNOTATED SLIDES

BROWSE COURSE MATERIAL

.toc { margin-left: 2em; } .lecslide { margin-top: 1em; margin-bottom: 1em; border-top: 0.5px solid #808080; padding-top: 1em; text-align: center; } .lecslideimg { width: 5in; border: 1px solid black; }

L09: Instruction Set Architectures

Content of the following slides is described in the surrounding text.

Welcome to Part 2 of 6.004! In this part of the course, we turn our attention to the design and implementation of digital systems that can perform useful computations on different types of binary

data. We'll come up with a general-purpose design for these systems, we which we call "computers", so that they can serve as useful tools in many diverse application areas. Computers were first used to perform numeric calculations in science and engineering, but today they are used as the central control element in any system where complex behavior is required.

Example: Factorial

factorial(N) = N! = N*(N-1)*…*1

C:

```
int a = 1;
int b = N;
do {
  a = a * b;
  b = b - 1;
} while (b != 0)
```

```
initially:    a =   1, b = 5
after iter 1: a =   5, b = 4
after iter 2: a =  20, b = 3
after iter 3: a =  60, b = 2
after iter 4: a = 120, b = 1
after iter 5: a = 120, b = 0
Done!
```

We have a lot to do in this lecture, so let's get started! Suppose we want to design a system to compute the factorial function on some numeric argument N. N! is defined as the product of N times N-1 times N-2, and so on down to 1.

We can use a programming language like C to describe the sequence of operations necessary to perform the factorial computation. In this program there are two variables, "a" and "b". "a" is used to accumulate the answer as we compute it step-by-step. "b" is used to hold the next value we need to multiply. "b" starts with the value of the numeric argument N. The DO loop is where the work gets done: on each loop iteration we perform one of the multiplies from the factorial formula, updating the value of the accumulator "a" with the result, then decrementing "b" in preparation for the next loop iteration.

Example: Factorial

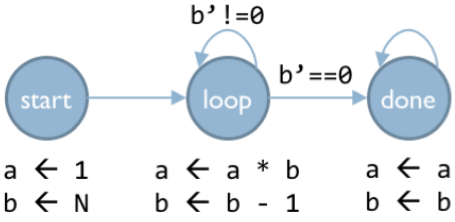
factorial(N) = N! = N*(N-1)*…*1

C:

```
int a = 1;
int b = N;
do {
  a = a * b;
  b = b - 1;
} while (b != 0)
```

```
start:  a ← 1,   b ← 5
loop:   a ← 5,   b ← 4
loop:   a ← 20,  b ← 3
loop:   a ← 60,  b ← 2
loop:   a ← 120, b ← 1
loop:   a ← 120, b ← 0
done:
```

High-level FSM:



- Helpful to translate into hardware
- D-registers (a, b)
- 2-bits of state (start, loop, done)
- Boolean transitions (b'==0, b'!=0)
- Register assignments in states

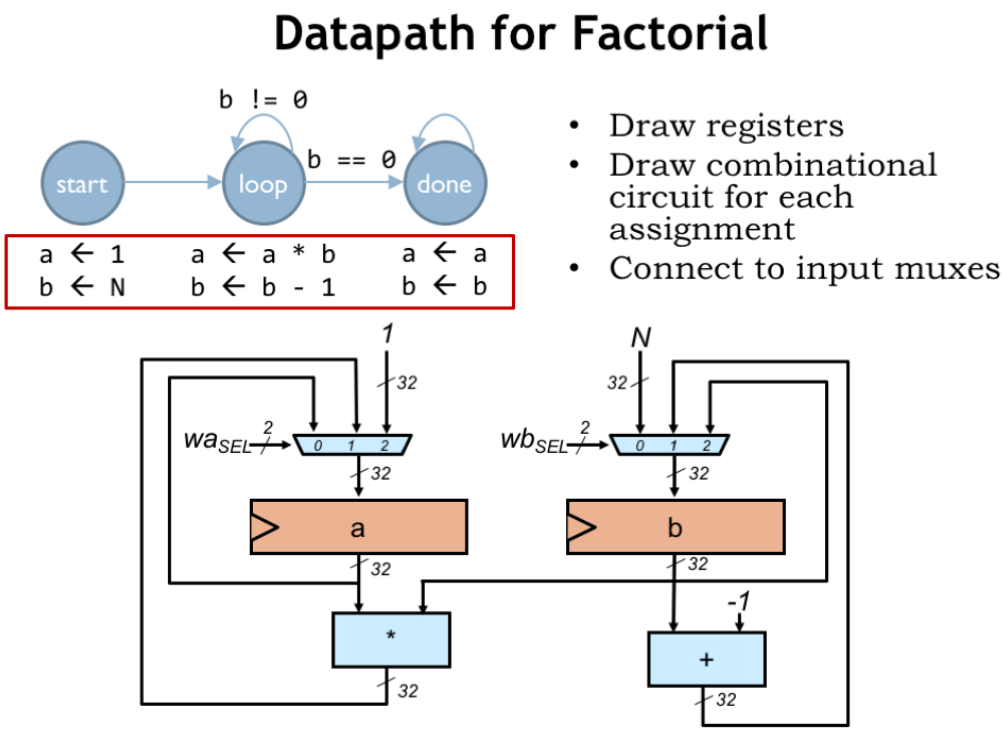
done:

(e.g., a ← a * b)

If we want to implement a digital system that performs this sequence of operations, it makes sense to use sequential logic! Here's the state transition diagram for a high-level finite-state machine designed to perform the necessary computations in the desired order. We call this a high-level FSM since the "outputs" of each state are more than simple logic levels. They are formulas indicating operations to be performed on source variables, storing the result in a destination variable.

The sequence of states visited while the FSM is running mirrors the steps performed by the execution of the C program. The FSM repeats the LOOP state until the new value to be stored in "b" is equal to 0, at which point the FSM transitions into the final DONE state.

The high-level FSM is useful when designing the circuitry necessary to implement the desired computation using our digital logic building blocks. We'll use 32-bit D-registers to hold the "a" and "b" values. And we'll need a 2-bit D-register to hold the 2-bit encoding of the current state, *i.e.*, the encoding for either START, LOOP or DONE. We'll include logic to compute the inputs required to implement the correct state transitions. In this case, we need to know if the new value for "b" is zero or not. And, finally, we'll need logic to perform multiply and decrement, and to select which value should be loaded into the "a" and "b" registers at the end of each FSM cycle.

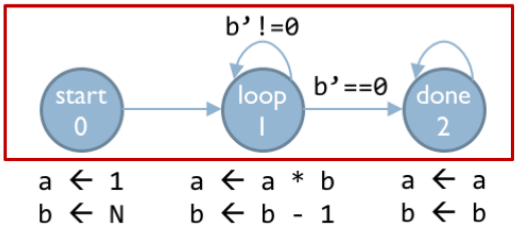


Let's start by designing the logic that implements the desired computations — we call this part of the logic the "datapath".

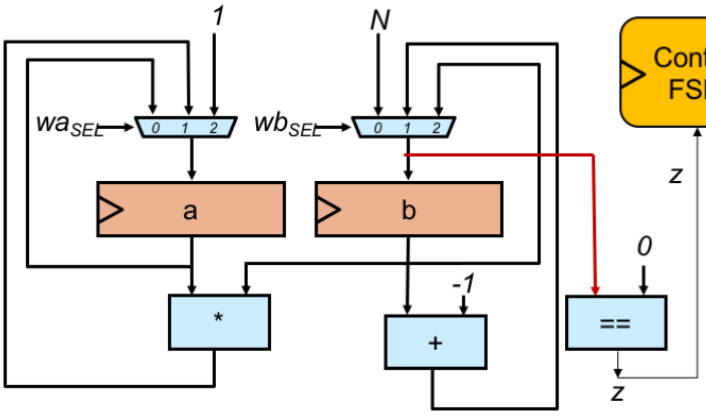
First we'll need two 32-bit D-registers to hold the "a" and "b" values. Then we'll draw the combinational logic blocks needed to compute the values to be stored in those registers. In the START state, we need the constant 1 to load into the "a" register and the constant N to load into the "b" register. In the LOOP state, we need to compute a*b for the "a" register and b-1 for the "b" register. Finally, in the DONE state, we need to be able to reload each register with its current value.

We'll use multiplexers to select the appropriate value to load into each of the data registers. These multiplexers are controlled by 2-bit select signals that choose which of the three 32-bit input values will be the 32-bit value to be loaded into the register. So by choosing the appropriate values for WASEL and WBSEL, we can make the datapath compute the desired values at each step in the FSM's operation.

Control FSM for Factorial



- Draw combinational logic for transition conditions
- Implement control FSM:
  - States: High-level FSM states
  - Inputs: Transition logic outputs
  - Outputs: Mux select signals

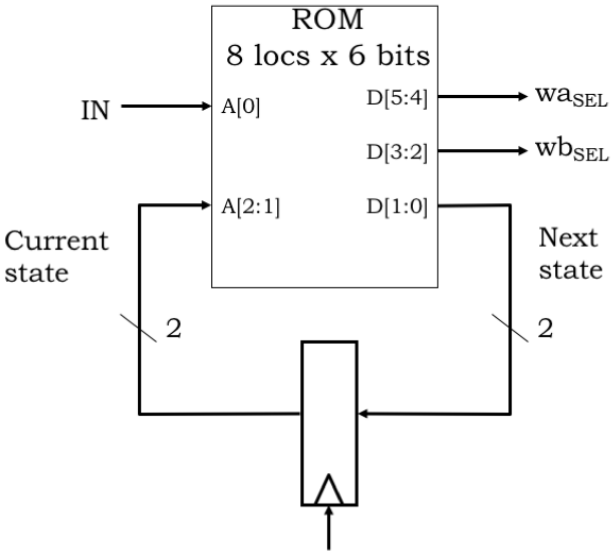


S	Z	wa_{SEL}	wb_{SEL}	S'
00	0	10	00	01
00	1	10	00	01
01	0	01	01	01
01	1	01	01	10
10	0	00	10	10
10	1	00	10	10

Next we'll add the combinational logic needed to control the FSM's state transitions. In this case, we need to test if the new value to be loaded into the "b" register is zero. The Z signal from the datapath will be 1 if that's the case and 0 otherwise.

Now we're all set to add the hardware for the control FSM, which has one input (Z) from the datapath and generates two 2-bit outputs (WASEL and WBSEL) to control the datapath. Here's the truth table for the FSM's combinational logic. S is the current state, encoded as a 2-bit value, and S' is the next state.

Control FSM Hardware



ROM contents

A[2:0]	D[5:0]
000	10 00 01
001	10 00 01
010	01 01 01
011	01 01 10
100	00 10 10
101	00 10 10

Using our skills from Part 1 of the course, we're ready to draw a schematic for the system! We know how to design the appropriate multiplier and decrement circuitry. And we can use our standard register-and-ROM implementation for the control FSM. The Z signal from the datapath is combined with the 2 bits of current state to form the 3 inputs to the combinational logic, in this case realized by a read-only memory with $2^3 = 8$ locations. Each ROM location has the appropriate values for the 6 output bits: 2 bits each for WASEL, WBSEL, and next state. The table on the right shows the ROM

output bits: 2 bits each for WASEL, WBSEL, and next state. The table on the right shows the ROM contents, which are easily determined from the table on the previous slide.

So Far: Single-Purpose Hardware

- Problem→ Procedure (High-level FSM)→ Implementation
- Systematic way to implement high-level FSM as a datapath + control FSM
  - Is this implementation an FSM itself?
  - If so, can you draw the truth table?
- How should we generalize our approach so we can solve many problems with one set of hardware?
  - More storage for operands and results
  - A larger repertoire of operations
  - General-purpose datapath

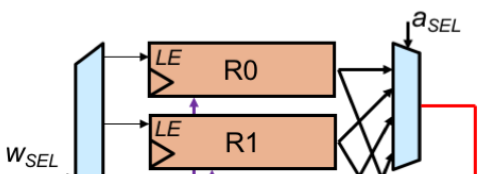
Okay, we've figured out a way to design hardware to perform a particular computation: Draw the state transition diagram for an FSM that describes the sequence of operations needed to complete the computation. Then construct the appropriate datapath, using registers to store values and combinational logic to implement the needed operations. Finally build an FSM to generate the control signals required by the datapath.

Is the datapath plus control logic itself an FSM? Well, it has registers and some combinational logic, so, yes, it is an FSM. Can we draw the truth table? In theory, yes. In practice, there are 66 bits of registers and hence 66 bits of state, so our truth table would need 2^{66} rows! Hmm, not very likely that we'd be able to draw the truth table! The difficulty comes from thinking of the registers in the datapath as part of the state of our super-FSM. That's why we think about the datapath as being separate from the control FSM.

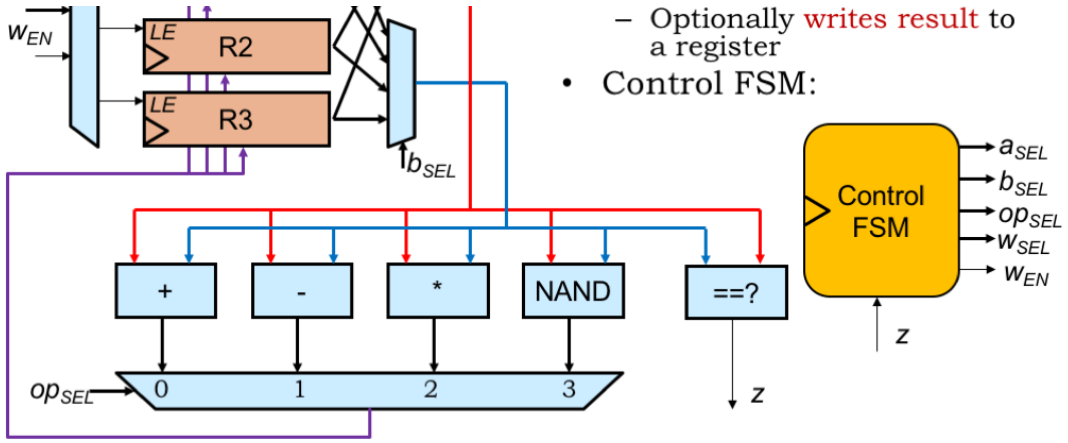
So how do we generalize this approach so we can use one computer circuit to solve many different problems? Well, most problems would probably require more storage for operands and results. And a larger list of allowable operations would be handy. This is actually a bit tricky: what's the minimum set of operations we can get away with? As we'll see later, surprisingly simple hardware is sufficient to perform any realizable computation. At the other extreme, many complex operations (*e.g.*, fast fourier transform) are best implemented as sequences of simpler operations (*e.g.*, add and multiply) rather than as a single massive combinational circuit. These sorts of design tradeoffs are what makes computer architecture fun!

We'd then combine our larger storage with logic for our chosen set of operations into a general purpose datapath that could be reused to solve many different problems. Let's see how that would work…

A Simple Programmable Datapath



- Each cycle, this datapath:
  - Reads two operands (a, b) from 4 registers (R0-R3)
  - Performs one operation of +, -, *, NAND on operands



Here's a datapath with 4 data registers to hold results. The ASEL and BSEL multiplexers allow any of the data registers to be selected as either operand for our repertoire of arithmetic and boolean operations. The result is selected by the OPSEL MUX and can be written back into any of the data registers by setting the WEN control signal to 1 and using the 2-bit WSEL signal to select which data register will be loaded at the next rising clock edge. Note that the data registers have a load-enable control input: when this signal is 1, the register will load a new value from its D input, otherwise it ignores the D input and simply reloads its previous value.

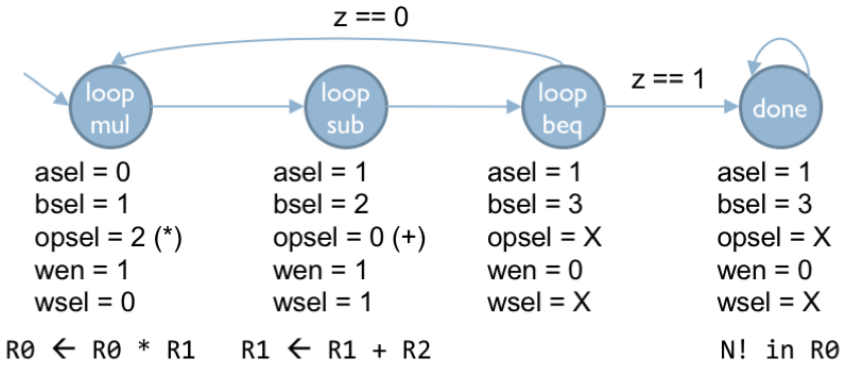
And, of course, we'll add a control FSM to generate the appropriate sequence of control signals for the datapath. The Z input from the datapath allows the system to perform data-dependent operations, where the sequence of operations can be influenced by the actual values in the data registers.

A Control FSM for Factorial

- Assume initial register contents:

```
R0 value = 1
R1 value = N
R2 value = -1
R3 value = 0
```

- Control FSM:



Here's the state transition diagram for the control FSM we'd use if we wanted to use this datapath to compute factorial assuming the initial contents of the data registers are as shown. We need a few more states than in our initial implementation since this datapath can only perform one operation at each step. So we need three steps for each iteration: one for the multiply, one for the decrement, and one for the test to see if we're done.

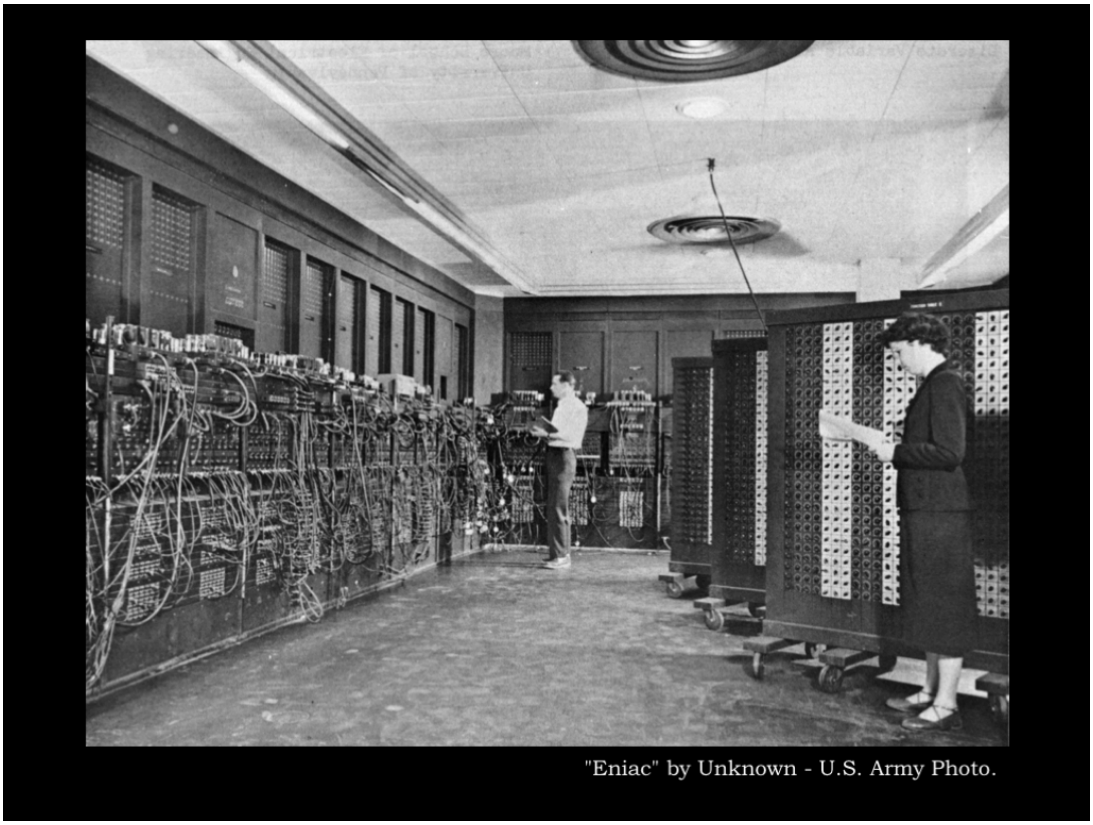
As seen here, it's often the case that general-purpose computer hardware will need more cycles and perhaps involve more hardware than an optimized single-purpose circuit.

New Problem → New Control FSM

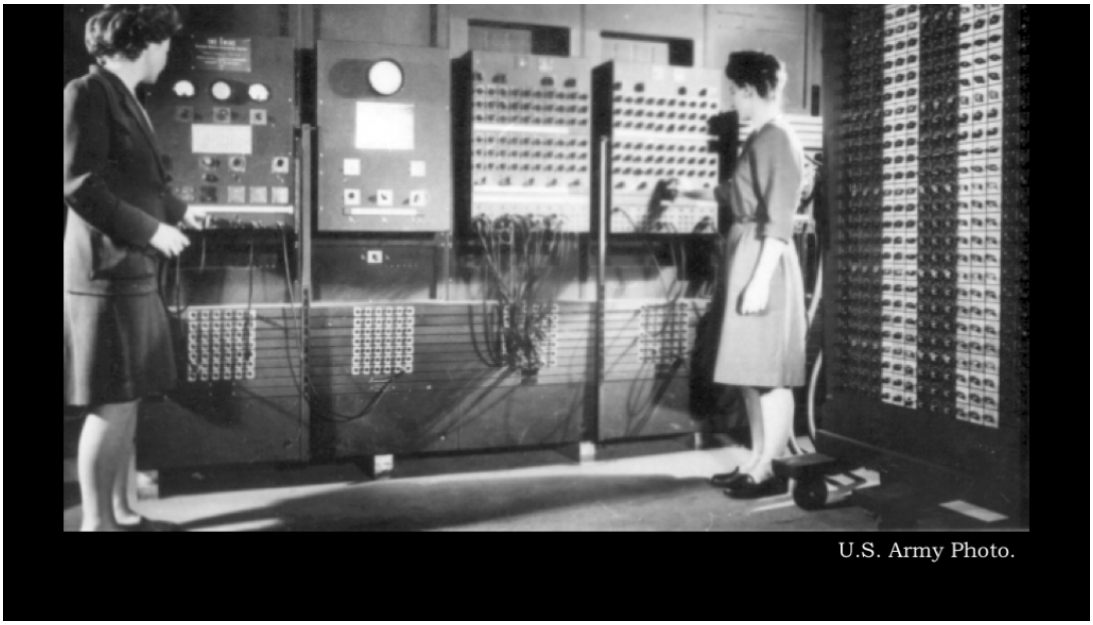
- You can solve many more problems with this datapath!
  - Exponentiation, division, square root, …
  - But nothing that requires more than four registers
- By designing a control FSM, we are programming the datapath
- Early digital computers were programmed this way!
  - ENIAC (1943):
    - First general-purpose digital computer
    - Programmed by setting huge array of dials and switches
    - Reprogramming it took about 3 weeks

You can solve many different problems with this system: exponentiation, division, square root, and so on, so long as you don't need more than four data registers to hold input data, intermediate results, or the final answer.

By designing a control FSM, we are in effect "programming" our digital system, specifying the sequence of operations it will perform.



This is exactly how the early digital computers worked! Here's a picture of the ENIAC computer built in 1943 at the University of Pennsylvania.

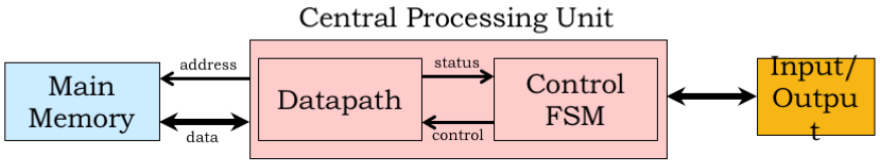


The Wikipedia article on the ENIAC tells us that “ENIAC could be programmed to perform complex sequences of operations, including loops, branches, and subroutines. The task of taking a problem and mapping it onto the machine was complex, and usually took weeks. After the program was figured out on paper, the process of getting the program into ENIAC by manipulating its switches and cables could take days. This was followed by a period of verification and debugging, aided by the ability to execute the program step by step.”

It’s clear that we need a less cumbersome way to program our computer!

The von Neumann Model

- Many approaches to build a general-purpose computer. Almost all modern computers are based on the von Neumann model (John von Neumann, 1945)
- Components:



- Central processing unit:
  Performs operations on values in registers & memory
- Main memory:
  Array of W words of N bits each
- Input/output devices to communicate with the outside world

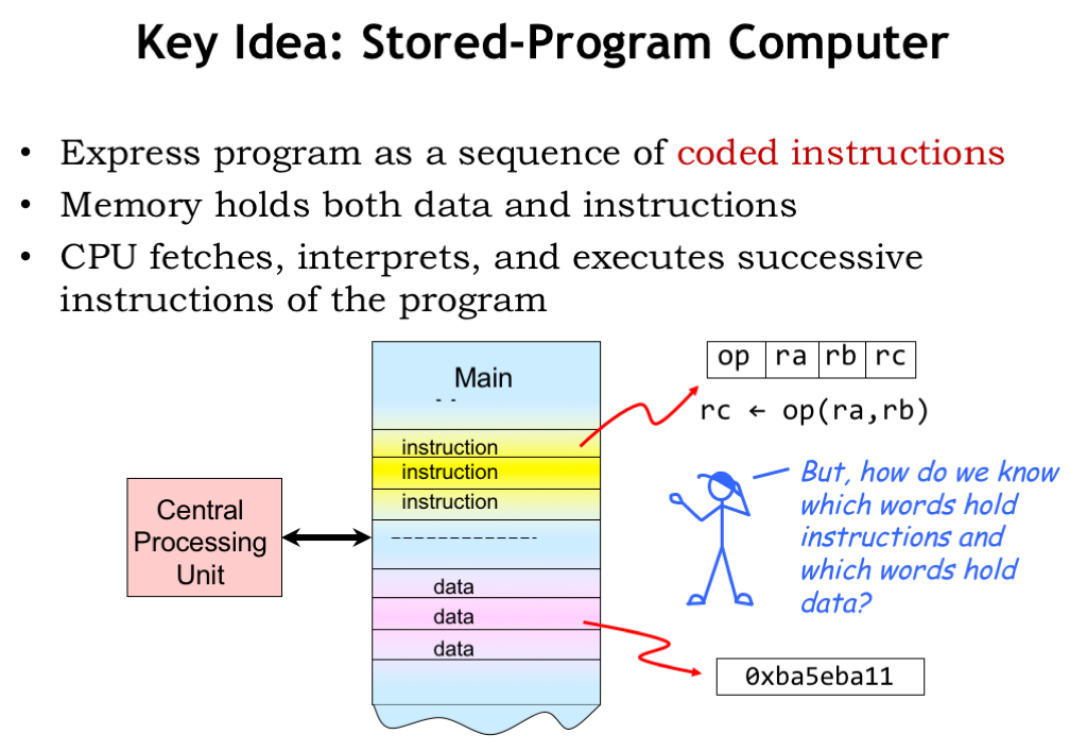
There are many approaches to building a general-purpose computer that can be easily re-programmed for new problems. Almost all modern computers are based on the “stored program” computer architecture developed by John von Neumann in 1945, which is now commonly referred to as the “von Neumann model”.

The von Neumann model has three components. There’s a central processing unit (aka the CPU) that contains a datapath and control FSM as described previously.

The CPU is connected to a read/write memory that holds some number W of words, each with N bits. Nowadays, even small memories have a billion words and the width of each location is at least 32 bits (usually more). This memory is often referred to as “main memory” to distinguish it from other

(usually more). This memory is often referred to as "main memory" to distinguish it from other memories in the system. You can think of it as an array: when the CPU wishes to operate on values in memory, it sends the memory an array index, which we call the address, and, after a short delay (currently 10's of nanoseconds) the memory will return the N-bit value stored at that address. Writes to main memory follow the same protocol except, of course, the data flows in the opposite direction. We'll talk about memory technologies a couple of lectures from now.

And, finally, there are input/output devices that enable the computer system to communicate with the outside world or to access data storage that, unlike main memory, will remember values even when turned off.

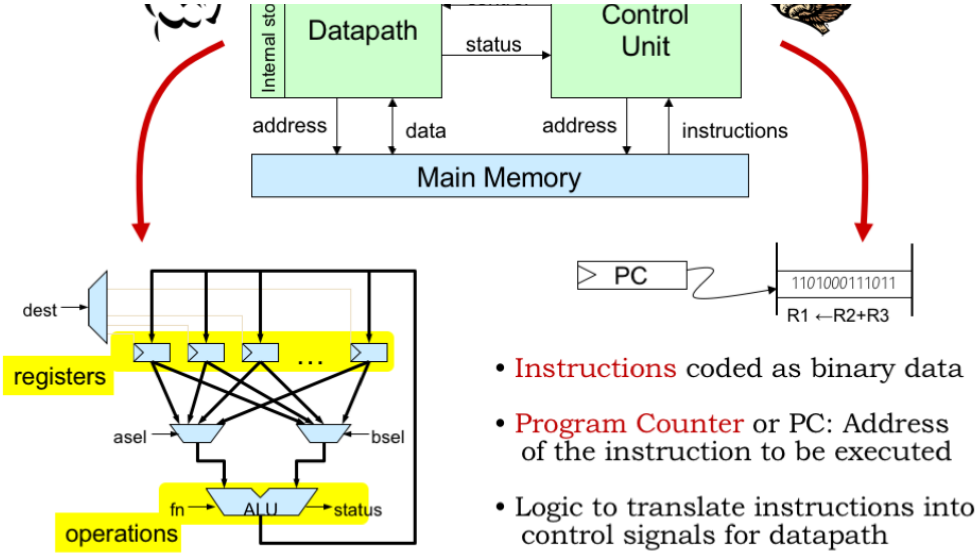


The key idea is to use main memory to hold the instructions for the CPU as well as data. Both instructions and data are, of course, just binary data stored in main memory.

Interpreted as an instruction, a value in memory can be thought of as a set of fields containing one or more bits encoding information about the actions to be performed by the CPU. The opcode field indicates the operation to be performed (*e.g.*, ADD, XOR, COMPARE). Subsequent fields specify which registers supply the source operands and the destination register where the result is stored. The CPU interprets the information in the instruction fields and performs the requested operation. It would then move on to the next instruction in memory, executing the stored program step-by-step. The goal of this chapter is to discuss the details of what operations we want the CPU to perform, how many registers we should have, and so on.

Of course, some values in memory are not instructions! They might be binary data representing numeric values, strings of characters, and so on. The CPU will read these values into its temporary registers when it needs to operate on them and write newly computed values back into memory.

Mr. Blue is asking a good question: how do we know which words in memory are instructions and which are data? After all, they're both binary values! The answer is that we can't tell by looking at the values — it's how they are used by the CPU that distinguishes instructions from data. If a value is loaded into the datapath, it's being used as data. If a value is loaded by the control logic, it's being used as an instruction.



So this is the digital system we'll build to perform computations. We'll start with a datapath that contains some number of registers to hold data values. We'll be able to select which registers will supply operands for the arithmetic and logic unit that will perform an operation. The ALU produces a result and other status signals. The ALU result can be written back to one of the registers for later use. We'll provide the datapath with means to move data to and from main memory.

There will be a control unit that provides the necessary control signals to the datapath. In the example datapath shown here, the control unit would provide ASEL and BSEL to select two register values as operands and DEST to select the register where the ALU result will be written. If the datapath had, say, 32 internal registers, ASEL, BSEL and DEST would be 5-bit values, each specifying a particular register number in the range 0 to 31. The control unit also provides the FN function code that controls the operation performed by the ALU. The ALU we designed in Part 1 of the course requires a 6-bit function code to select between a variety of arithmetic, boolean and shift operations.

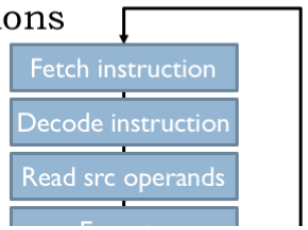
The control unit would load values from main memory to be interpreted as instructions. The control unit contains a register, called the "program counter", that keeps track of the address in main memory of the next instruction to be executed. The control unit also contains a (hopefully small) amount of logic to translate the instruction fields into the necessary control signals. Note the control unit receives status signals from the datapath that will enable programs to execute different sequences of instructions if, for example, a particular data value was zero.

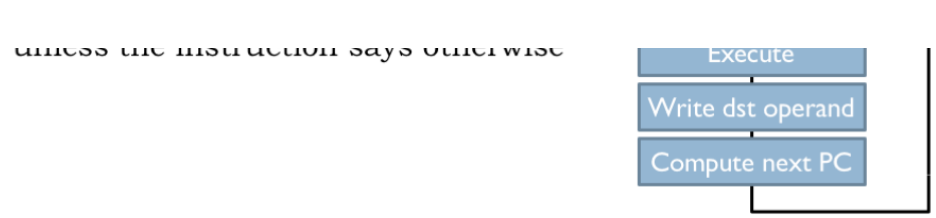
The datapath serves as the brawn of our digital system and is responsible for storing and manipulating data values. The control unit serves as the brain of our system, interpreting the program stored in main memory and generating the necessary sequence of control signals for the datapath.

Instructions

- Instructions are the fundamental unit of work
- Each instruction specifies:
  - An operation or opcode to be performed
  - Source operands and destination for the result

- In a von Neumann machine, instructions are executed sequentially
  - CPU logically implements this loop:
  - By default, the next PC is current PC + size of current instruction unless the instruction says otherwise





Instructions are the fundamental unit of work. They're fetched by the control unit and executed one after another in the order they are fetched. Each instruction specifies the operation to be performed along with the registers to supply the source operands and destination register where the result will be stored.

In a von Neumann machine, instruction execution involves the steps shown here: the instruction is loaded from the memory location whose address is specified by the program counter. When the requested data is returned by the memory, the instruction fields are converted to the appropriate control signals for the datapath, selecting the source operands from the specified registers, directing the ALU to perform the specified operation, and storing the result in the specified destination register. The final step in executing an instruction is updating the value of the program counter to be the address of the next instruction.

This execution loop is performed again and again. Modern machines can execute more than a billion instructions per second!

Instruction Set Architecture (ISA)

- ISA: The contract between software and hardware
  - Functional definition of operations and storage locations
  - Precise description of how software can invoke and access them
- The ISA is a new layer of abstraction:
  - ISA specifies what the hardware provides, not how it's implemented
  - Hides the complexity of CPU implementation
  - Enables fast innovation in hardware (no need to change software!)
    - 8086 (1978): 29 thousand transistors, 5 MHz, 0.33 MIPS
    - Pentium 4 (2003): 44 million transistors, 4 GHz, ~5000 MIPS
    - Both implement x86 ISA
  - Dark side: Commercially successful ISAs last for decades
    - Today's x86 CPUs carry baggage of design decisions from the 70's

The discussion so far has been a bit abstract. Now it's time to roll up our sleeves and figure out what instructions we want our system to support. The specification of instruction fields and their meaning along with the details of the datapath design are collectively called the instruction set architecture (ISA) of the system. The ISA is a detailed functional specification of the operations and storage mechanisms and serves as a contract between the designers of the digital hardware and the programmers who will write the programs. Since the programs are stored in main memory and can hence be changed, we'll call them software, to distinguish them from the digital logic which, once implemented, doesn't change. It's the combination of hardware and software that determine the behavior of our system.

The ISA is a new layer of abstraction: we can write programs for the system without knowing the implementation details of the hardware. As hardware technology improves we can build faster systems without having to change the software. You can see here that over a fifteen year timespan, the hardware for executing the Intel x86 instruction set went from executing 300,000 instructions per second to executing 5 billion instructions per second. Same software as before, we've just taken

second to executing 3 billion instructions per second. Same software as before, we've just taken advantage of smaller and faster MOSFETs to build more complex circuits and faster execution engines.

But a word of caution is in order! It's tempting to make choices in the ISA that reflect the constraints of current technologies, *e.g.*, the number of internal registers, the width of the operands, or the maximum size of main memory. But it will be hard to change the ISA when technology improves since there's a powerful economic incentive to ensure that old software can run on new machines, which means that a particular ISA can live for decades and span many generations of technology. If your ISA is successful, you'll have to live with any bad choices you made for a very long time.

Instruction Set Architecture Design

- Designing an ISA is hard:
  - How many operations?
  - What types of storage, how much?
  - How to encode instructions?
  - How to future-proof?
- How to decide? Take a quantitative approach
  - Take a set of representative benchmark programs
  - Evaluate versions of your ISA and implementation with and without feature
  - Pick what works best overall (performance, energy, area…)
- Corollary: Optimize the common case

Let's design our own instruction set: the Beta!

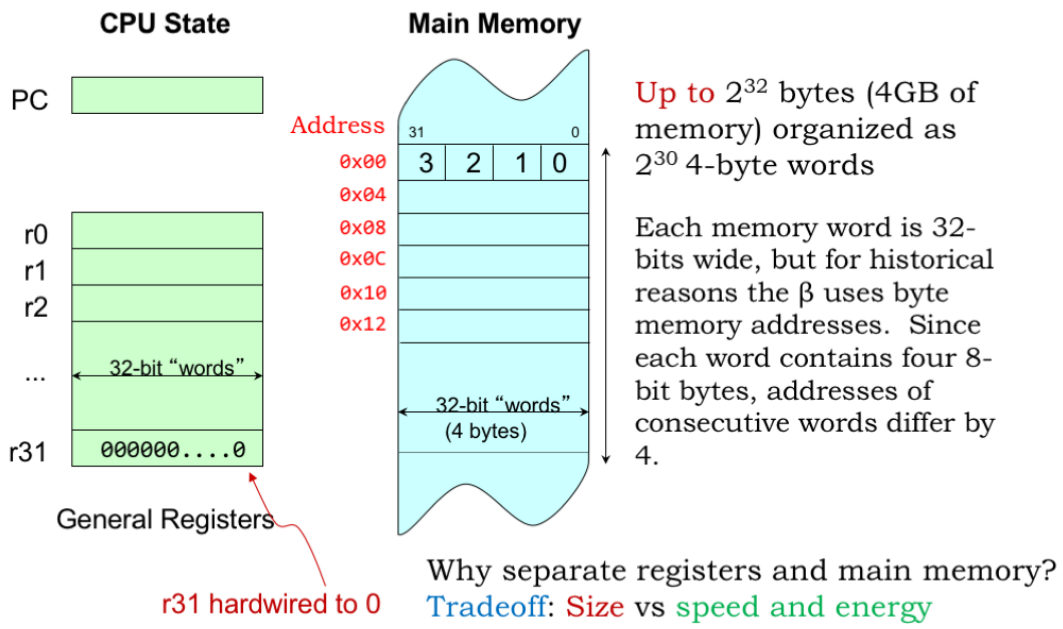
Designing an ISA is hard! What are the operations that should be supported? How many internal registers? How much main memory? Should we design the instruction encoding to minimize program size or to keep the logic in the control unit as simple as possible? Looking into our crystal ball, what can we say about the computation and storage capabilities of future technologies?

We'll answer these questions by taking a quantitative approach. First we'll choose a set of benchmark programs, chosen as representative of the many types of programs we expect to run on our system. So some benchmark programs will perform scientific and engineering computations, some will manipulate large data sets or perform database operations, some will require specialized computations for graphics or communications, and so on. Happily, after many decades of computer use, several standardized benchmark suites are available for us to use.

We'll then implement the benchmark programs using our instruction set and simulate their execution on our proposed datapath. We'll evaluate the results to measure how well the system performs. But what do we mean by "well"? That's where it gets interesting: "well" could refer to execution speed, energy consumption, circuit size, system cost, etc. If you're designing a smart watch, you'll make different choices than if you're designing a high-performance graphics card or a data-center server.

Whatever metric you choose to evaluate your proposed system, there's an important design principle we can follow: identify the common operations and focus on them as you optimize your design. For example, in general-purpose computing, almost all programs spend a lot of their time on simple arithmetic operations and accessing values in main memory. So those operations should be made as fast and energy efficient as possible.

Now, let's get to work designing our own instruction set and execution engine, a system we'll call the Beta.



The Beta is an example of a reduced-instruction-set computer (RISC) architecture. "Reduced" refers to the fact that in the Beta ISA, most instructions only access the internal registers for their operands and destination. Memory values are loaded and stored using separate memory-access instructions, which implement only a simple address calculation. These reductions lead to smaller, higher-performance hardware implementations and simpler compilers on the software side. The ARM and MIPS ISAs are other examples of RISC architectures. Intel's x86 ISA is more complex.

There is a limited amount of storage inside of the CPU — using the language of sequential logic, we'll refer to this as the CPU state. There's a 32-bit program counter (PC for short) that holds the address of the current instruction in main memory. And there are thirty-two registers, numbered 0 through 31. Each register holds a 32-bit value. We'll use use 5-bit fields in the instruction to specify the number of the register to be used an operand or destination. As shorthand, we'll refer to a register using the prefix "R" followed by its number, *e.g.*, "R0" refers to the register selected by the 5-bit field 0b00000.

Register 31 (R31) is special — its value always reads as 0 and writes to R31 have no affect on its value.

The number of bits in each register and hence the number of bits supported by ALU operations is a fundamental parameter of the ISA. The Beta is a 32-bit architecture. Many modern computers are 64-bit architectures, meaning they have 64-bit registers and a 64-bit datapath.

Main memory is an array of 32-bit words. Each word contains four 8-bit bytes. The bytes are numbered 0 through 3, with byte 0 corresponding to the low-order 7 bits of the 32-bit value, and so on. The Beta ISA only supports word accesses, either loading or storing full 32-bit words. Most "real" computers also support accesses to bytes and half-words.

Even though the Beta only accesses full words, following a convention used by many ISAs it uses byte addresses. Since there are 4 bytes in each word, consecutive words in memory have addresses that differ by 4. So the first word in memory has address 0, the second word address 4, and so on. You can see the addresses to left of each memory location in the diagram shown here. Note that we'll usually use hexadecimal notation when specifying addresses and other binary values — the "0x" prefix indicates when a number is in hex. When drawing a memory diagram, we'll follow the convention that addresses increase as you read from top to bottom.

The Beta ISA supports 32-bit byte addressing, so an address fits exactly into one 32-bit register or memory location. The maximum memory size is 2^{32} bytes or 2^{30} words — that's 4 gigabytes (4 GB) or one billion words of main memory. Some Beta implementations might actually have a smaller main memory, *i.e.*, one with fewer than 1 billion locations.

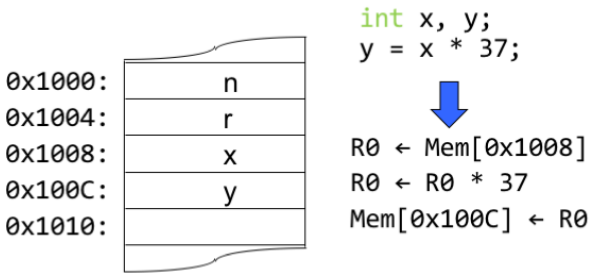
memory, i.e., one with fewer than 4 billion locations.

Why have separate registers and main memory? Well, modern programs and datasets are very large, so we'll want to have a large main memory to hold everything. But large memories are slow and usually only support access to one location at a time, so they don't make good storage for use in each instruction which needs to access several operands and store a result. If we used only one large storage array, then an instruction would need to have three 32-bit addresses to specify the two source operands and destination — each instruction encoding would be huge! And the required memory accesses would have to be one-after-the-other, really slowing down instruction execution.

On the other hand, if we use registers to hold the operands and serve as the destination, we can design the register hardware for parallel access and make it very fast. To keep the speed up we won't be able to have very many registers — a classic size-vs-speed performance tradeoff we see in digital systems all the time. In the end, the tradeoff leading to the best performance is to have a small number of very fast registers used by most instructions and a large but slow main memory. So that's what the BETA ISA does.

Storage Conventions

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
  - Load them
  - Compute on them
  - Store the results



In general, all program data will reside in main memory. Each variable used by the program "lives" in a specific main memory location and so has a specific memory address. For example, in the diagram below, the value of variable "x" is stored in memory location 0x1008, and the value of "y" is stored in memory location 0x100C, and so on.

To perform a computation, *e.g.*, to compute x*37 and store the result in y, we would have to first load the value of x into a register, say, R0. Then we would have the datapath multiply the value in R0 by 37, storing the result back into R0. Here we've assumed that the constant 37 is somehow available to the datapath and doesn't itself need to be loaded from memory. Finally, we would write the updated value in R0 back into memory at the location for y.

Whew! A lot of steps… Of course, we could avoid all the loading and storing if we chose to keep the values for x and y in registers. Since there are only 32 registers, we can't do this for all of our variables, but maybe we could arrange to load x and y into registers, do all the required computations involving x and y by referring to those registers, and then, when we're done, store changes to x and y back into memory for later use. Optimizing performance by keeping often-used values in registers is a favorite trick of programmers and compiler writers.

So the basic program template is some loads to bring values into the registers, followed by computation, followed by any necessary stores. ISAs that use this template are usually referred to as "load-store architectures".

Beta ISA: Instructions

- Three types of instructions:
  - Arithmetic and logical: Perform operations on general registers
  - Loads and stores: Move data between general registers and main memory
  - Branches: Conditionally change the program counter

- All instructions have a fixed length: 32 bits (4 bytes)
  - Tradeoff (vs variable-length instructions):
    - Simpler decoding logic, next PC is easy to compute
    - Larger code size

Having talked about the storage resources provided by the Beta ISA, let's design the Beta instructions themselves. This might be a good time to print a copy of the handout called the "Summary of Beta Instruction Formats" so you'll have it for handy reference.

The Beta has three types of instructions: compute instructions that perform arithmetic and logic operations on register values, load and store instructions that access values in main memory, and branch instructions that change the value of the program counter.

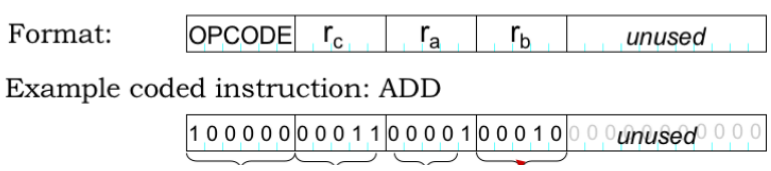
We'll discuss each class of instructions in turn.

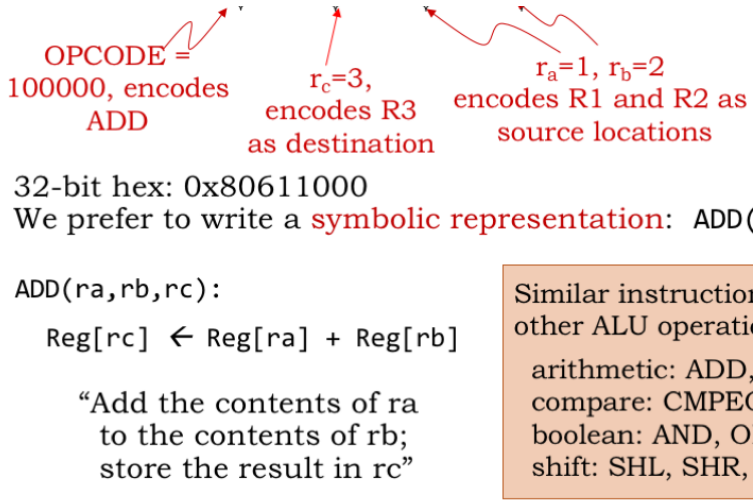
In the Beta ISA, all the instruction encodings are the same size: each instruction is encoded in 32 bits and hence occupies exactly one 32-bit word in main memory. This instruction encoding leads to simpler control-unit logic for decoding instructions. And computing the next value of the program counter is very simple: for most instructions, the next instruction can be found in the following memory location. We just need to add 4 to the current value of program counter to advance to the next instruction.

As we saw in Part 1 of the course, fixed-length encodings are often inefficient in the sense that the same information content (in this case, the encoded program) can be encoded using fewer bits. To do better we would need a variable-length encoding for instructions, where frequently-occurring instructions would use a shorter encoding. But hardware to decode variable-length instructions is complex since there may be several instructions packed into one memory word, while other instructions might require loading several memory words. The details can be worked out, but there's a performance and energy cost associated with the more efficient encoding.

Nowadays, advances in memory technology have made memory size less of an issue and the focus is on the higher-performance needed by today's applications. Our choice of a fixed-length encoding leads to larger code size, but keeps the hardware execution engine small and fast.

Beta ALU Instructions





The computation performed by the Beta datapath happens in the arithmetic-and-logic unit (ALU). We'll be using the ALU designed in Part 1 of the course.

The Beta ALU instructions have 4 instruction fields. There's a 6-bit field specifying the ALU operation to be performed — this field is called the opcode. The two source operands come from registers whose numbers are specified by the 5-bit "ra" and "rb" fields. So we can specify any register from R0 to R31 as a source operand. The destination register is specified by the 5-bit "rc" field.

This instruction format uses 21 bits of the 32-bit word, the remaining bits are unused and should be set to 0. The diagram shows how the fields are positioned in the 32-bit word. The choice of position for each field is somewhat arbitrary, but to keep the hardware simple, when we can we'll want to use the same field positions for similar fields in the other instruction encodings. For example, the opcode will always be found in bits [31:26] of the instruction.

Here's the binary encoding of an ADD instruction. The opcode for ADD is the 6-bit binary value 0b100000 — you can find the binary for each opcode in the Opcode Table in the handout mentioned before. The "rc" field specifies that the result of the ADD will be written into R3. And the "ra" and "rb" fields specify that the first and second source operands are R1 and R2 respectively. So this instruction adds the 32-bit values found in R1 and R2, writing the 32-bit sum into R3.

Note that it's permissible to refer to a particular register several times in the same instruction. So, for example, we could specify R1 as the register for both source operands AND also as the destination register. If we did, we'd be adding R1 to R1 and writing the result back into R1, which would effectively multiply the value in R1 by 2.

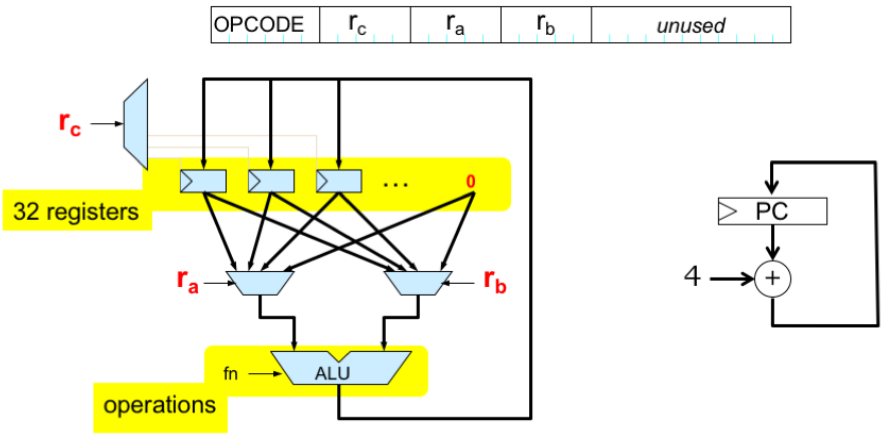
Since it's tedious and error-prone to transcribe 32-bit binary values, we'll often use hexadecimal notation for the binary representation of an instruction. In this example, the hexadecimal notation for the encoded instruction is 0x80611000. However, it's *much* easier if we describe the instructions using a functional notation, *e.g.*, "ADD(r1,r2,r3)". Here we use a symbolic name for each operation, called a mnemonic. For this instruction the mnemonic is "ADD", followed by a parenthesized list of operands, in this case the two source operands (r1 and r2), then the destination (r3). So we'll understand that ADD(ra,rb,rc) is shorthand for asking the Beta to compute the sum of the values in registers ra and rb, writing the result as the new value of register rc.

Here's the list of the mnemonics for all the operations supported by the Beta. There is a detailed description of what each instruction does in the Beta Documentation handout. Note that all these instructions use same 4-field template, differing only in the value of the opcode field. This first step was pretty straightforward — we simply provided instruction encodings for the basic operations provided by the ALU.

Implementation Sketch #1

Now that we have our first set of instructions, we can create a

more concrete implementation sketch:



Now that we have our first group of instructions, we can create a more concrete implementation sketch.

Here we see our proposed datapath. The 5-bit “ra” and “rb” fields from the instruction are used to select which of the 32 registers will be used for the two operands. Note that register 31 isn’t actually a read/write register, it’s just the 32-bit constant 0, so that selecting R31 as an operand results in using the value 0. The 5-bit “rc” field from the instruction selects which register will be written with the result from the ALU. Not shown is the hardware needed to translate the instruction opcode to the appropriate ALU function code — perhaps a 64-location ROM could be used to perform the translation by table lookup.

The program counter logic supports simple sequential execution of instructions. It’s a 32-bit register whose value is updated at the end of each instruction by adding 4 to its current value. This means the next instruction will come from the memory location following the one that holds the current instruction.

In this diagram we see one of the benefits of a RISC architecture: there’s not much logic needed to decode the instruction to produce the signals needed to control the datapath. In fact, many of the instruction fields are used as-is!

Should We Support Constant Operands?

Many programs use small constants frequently

e.g., our factorial example: 0, 1, -1

Tradeoff:

When used, they save registers and instructions

More opcodes → more complex control logic and datapath

Analyzing operands when running SPEC CPU benchmarks, we find that constant operands appear in

- >50% of executed arithmetic instructions
  - *Loop increments, scaling indicies*
- >80% of executed compare instructions
  - *Loop termination condition*
- >25% of executed load instructions
  - *Offsets into data structures*

ISA designers receive many requests for what are affectionately known as “features” — additional instructions that, in theory, will make the ISA better in some way. Dealing with such requests is the moment to apply our quantitative approach in order to be able to judge the tradeoffs between cost and benefits.

Our first “feature request” is to allow small constants as the second operand in ALU instructions. So if we replaced the 5-bit “rb” field, we would have room in the instruction to include a 16-bit constant as bits [15:0] of the instruction. The argument in favor of this request is that small constants appear frequently in many programs and it would make programs shorter if we didn’t have use load operations to read constant values from main memory. The argument against the request is that we would need additional control and datapath logic to implement the feature, increasing the hardware cost and probably decreasing the performance.

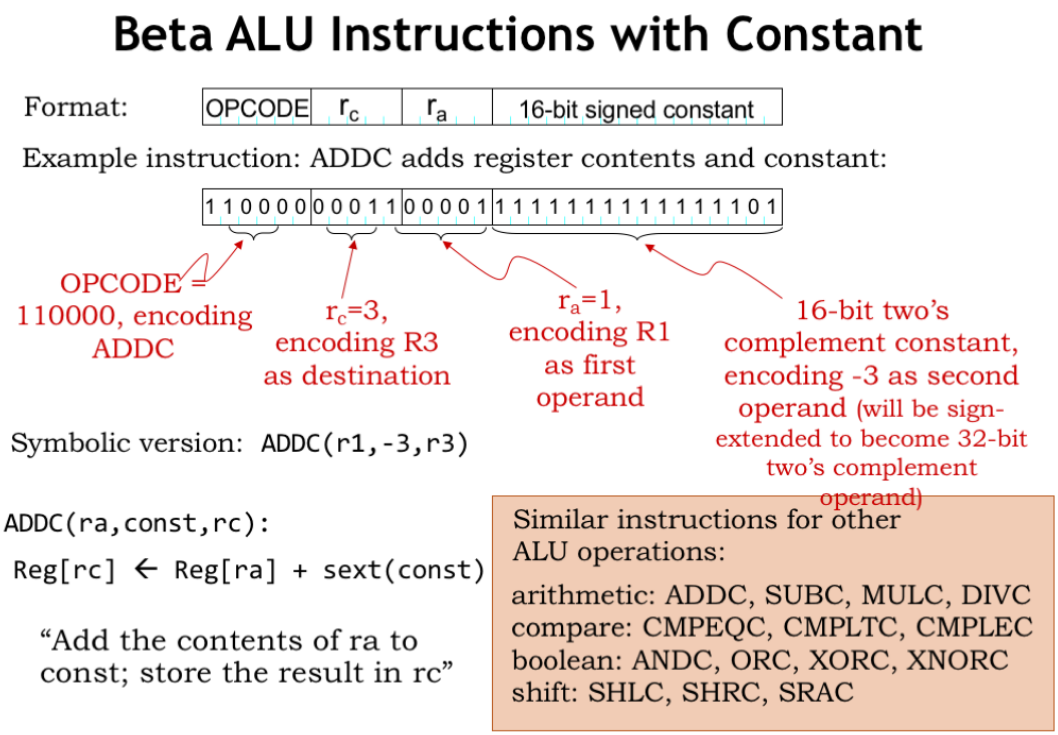
So our strategy is to modify our benchmark programs to use the ISA augmented with this feature and measure the impact on a simulated execution. Looking at the results, we find that there is compelling evidence that small constants are indeed very common as the second operands to many operations. Note that we’re not so much interested in simply looking at the code. Instead we want to look at what instructions actually get executed while running the benchmark programs. This will take into account that instructions executed during each iteration of a loop might get executed 1000’s of times even though they only appear in the program once.

Looking at the results, we see that over half of the arithmetic instructions have a small constant as their second operand.

Comparisons involve small constants 80% of the time. This probably reflects the fact that during execution comparisons are used in determining whether we’ve reached the end of a loop.

And small constants are often found in address calculations done by load and store operations.

Operations involving constant operands are clearly a common case, one well worth optimizing. Adding support for small constant operands to the ISA resulted in programs that were measurably smaller and faster. So: feature request approved!



Here we see the second of the two Beta instruction formats. It’s a modification of the first format where we’ve replaced the 5-bit “rb” field with a 16-bit field holding a constant in two’s complement format. This will allow us to represent constant operands in the range of 0x8000 (decimal -32768) to 0x7FFF (decimal 32767).

Here's an example of the add-constant (ADDC) instruction which adds the contents of R1 and the constant -3, writing the result into R3. We can see that the second operand in the symbolic representation is now a constant (or, more generally, an expression that can evaluated to get a constant value).

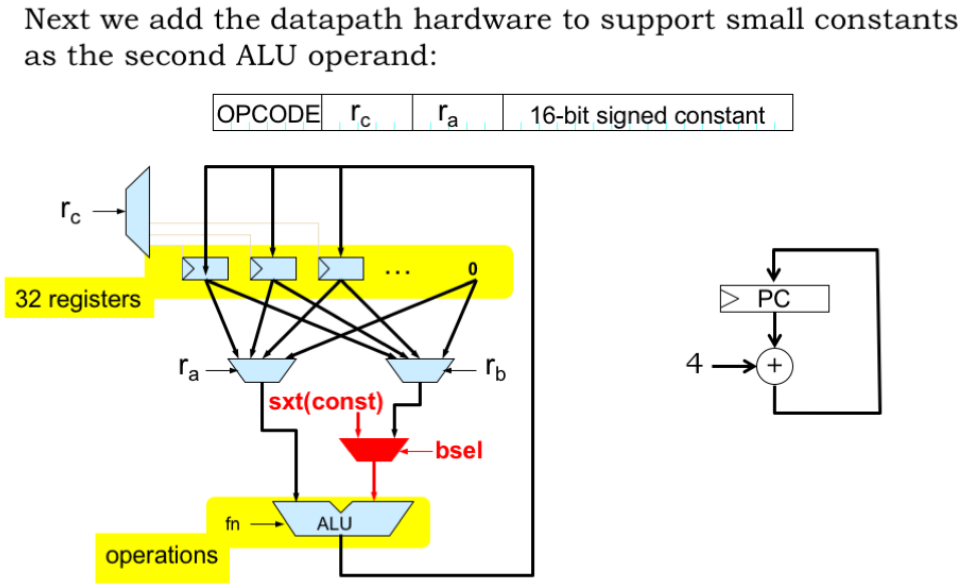
One technical detail needs discussion: the instruction contains a 16-bit constant, but the datapath requires a 32-bit operand. How does the datapath hardware go about converting from, say, the 16-bit representation of -3 to the 32-bit representation of -3?

Comparing the 16-bit and 32-bit representations for various constants, we see that if the 16-bit two's-complement constant is negative (*i.e.*, its high-order bit is 1), the high sixteen bits of the equivalent 32-bit constant are all 1's. And if the 16-bit constant is non-negative (*i.e.*, its high-order bit is 0), the high sixteen bits of the 32-bit constant are all 0's. Thus the operation the hardware needs to perform is "sign extension" where the sign-bit of the 16-bit constant is replicated sixteen times to form the high half of the 32-constant. The low half of the 32-bit constant is simply the 16-bit constant from the instruction. No additional logic gates will be needed to implement sign extension — we can do it all with wiring.

Here are the fourteen ALU instructions in their "with constant" form, showing the same instruction mnemonics but with a "C" suffix indicate the second operand is a constant. Since these are additional instructions, these have different opcodes than the original ALU instructions.

Finally, note that if we need a constant operand whose representation does NOT fit into 16 bits, then we have to store the constant as a 32-bit value in a main memory location and load it into a register for use just like we would any variable value.

Implementation Sketch #2

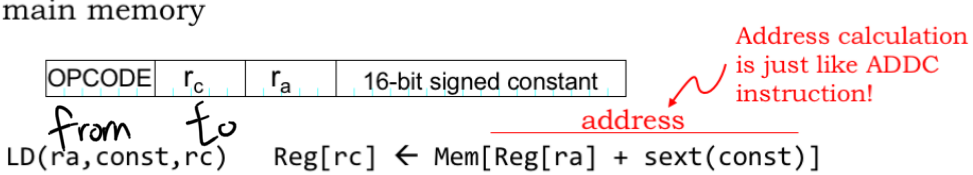


To give some sense for the additional datapath hardware that will be needed, let's update our implementation sketch to add support for constants as the second ALU operand. We don't have to add much hardware: just a multiplexer which selects either the "rb" register value or the sign-extended constant from the 16-bit field in the instruction. The BSEL control signal that controls the multiplexer is 1 for the ALU-with-constant instructions and 0 for the regular ALU instructions.

We'll put the hardware implementation details aside for now and revisit them in a few lectures.

Beta Load and Store Instructions

Loads and stores move data between the internal registers and main memory




```
LD(ra,const,rc)    Reg[rc] ← Mem[Reg[ra] + sext(const)]
```

Load rc with the contents of the memory location

```
ST(rc,const,ra)    Mem[Reg[ra] + sext(const)] ← Reg[rc]
```

Store the contents of rc into the memory location

To access memory the CPU has to generate an address. LD and ST compute the address by adding the sign-extended constant to the contents of register ra.

- To access a constant address, specify R31 as ra.
- To use only a register value as the address, specify a constant of 0.

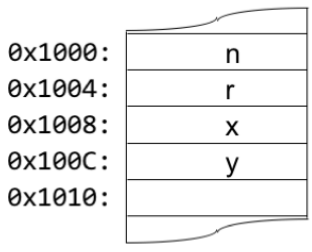
Now let's turn our attention to the second class of instructions: load (LD) and store (ST), which allow the CPU to access values in memory. Note that since the Beta is a load-store architecture these instructions are the *only* mechanism for accessing memory values.

The LD and ST instructions use the same instruction template as the ALU-with-constant instructions. To access memory, we'll need a memory address, which is computed by adding the value of the "ra" register to the sign-extended 16-bit constant from the low-order 16 bits of the instruction. This computation is exactly the one performed by the ADDC instruction — so we'll reuse that hardware — and the sum is sent to main memory as the byte address of the location to be accessed. For the LD instruction, the data returned by main memory is written to the "rc" register.

The store instruction (ST) performs the same address calculation as LD, then reads the data value from the "rc" register and sends both to main memory. The ST instruction is special in several ways: it's the only instruction that needs to read the value of the "rc" register, so we'll need to adjust the datapath hardware slightly to accommodate that need. And since "rc" is serving as a source operand, it appears as the first operand in the symbolic form of the instruction, followed by "const" and "ra" which are specifying the destination address. ST is the only instruction that does *not* write a result into the register file at end of the instruction.

Using LD and ST

- Variables live in memory
- Registers hold temporary values
- To operate with memory variables
  - Load them
  - Compute on them
  - Store the results




```
int x, y;
y = x * 37;
```

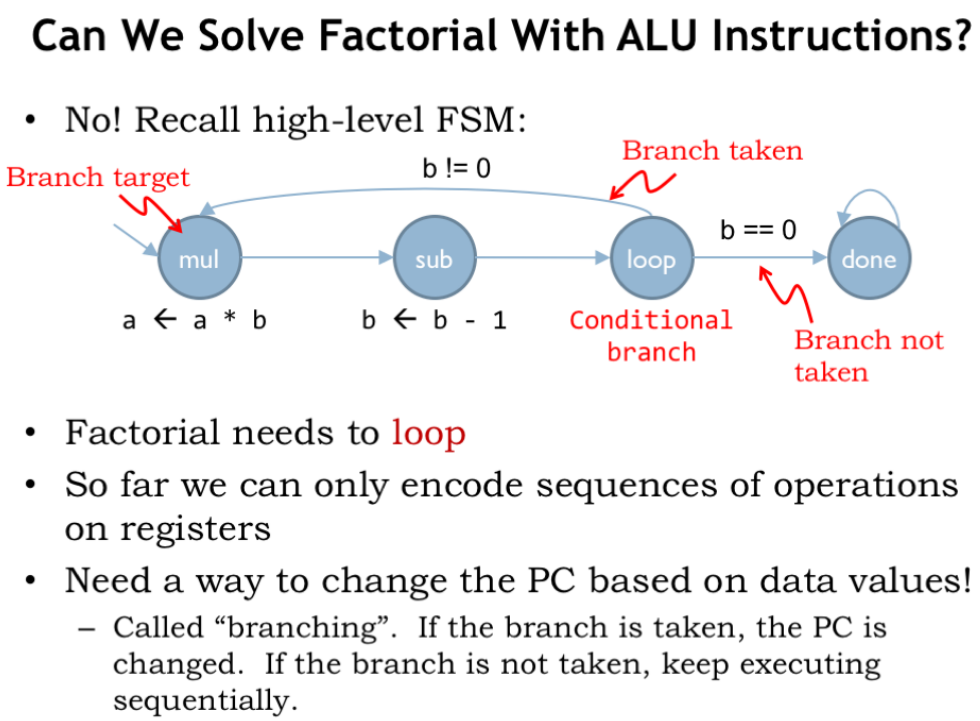
```
R0 ← Mem[0x1008]
R0 ← R0 * 37
Mem[0x100C] ← R0
```

```
LD(R31,0x1008,R0)
MULC(R0,37,R0)
ST(R0,0x100C,R31)
```

Here's the example we saw earlier, where we needed to load the value of the variable x from memory, multiply it by 37 and write the result back to the memory location that holds the value of the variable y.

Now that we have actual Beta instructions, we've expressed the computation as a sequence of three instructions. To access the value of variable x, the LD instruction adds the contents of R31 to the constant 0x1008, which sums to 0x1008, the address we need to access. The ST instruction specifies a similar address calculation to write into the location for the variable y.

The address calculation performed by LD and ST works well when the locations we need to access have addresses that fit into the 16-bit constant field. What happens when we need to access locations at addresses higher than 0x7FFF? Then we need to treat those addresses as we would any large constant, and store those large addresses in main memory so they can be loaded into a register to be used by LD and ST. Okay, but what if the number of large constants we need to store is greater than will fit in low memory, *i.e.*, the addresses we can access directly? To solve this problem, the Beta includes a "load relative" (LDR) instruction, which we'll see in the lecture on the Beta implementation.



Finally, let's discuss the third class of instructions that let us change the program counter. Up until now, the program counter has simply been incremented by 4 at the end of each instruction, so that the next instruction comes from the memory location that immediately follows the location that held the current instruction, *i.e.*, the Beta has been executing instructions sequentially from memory.

But in many programs, such as in factorial, we need to disrupt sequential execution, either to loop back to repeat some earlier instruction, or to skip over instructions because of some data dependency. We need a way to change the program counter based on data values generated by the program's execution. In the factorial example, as long as b is not equal to 0, we need to keep executing the instructions that calculate a*b and decrement b. So we need instructions to test the value of b after it's been decremented and if it's non-zero, change the PC to repeat the loop one more time.

Changing the PC depending on some condition is implemented by a branch instruction, and the operation is referred to as a "conditional branch". When the branch is taken, the PC is changed and execution is restarted at the new location, which is called the branch target. If the branch is not taken, the PC is incremented by 4 and execution continues with the instruction following the branch.

As the name implies, a branch instruction represents a potential fork in the execution sequence. We'll

use branches to implement many different types of control structures: loops, conditionals, procedure calls, etc.

Beta Branch Instructions

The Beta's *branch instructions* provide a way to conditionally change the PC to point to a nearby location…

… and, optionally, remembering (in Rc) where we came from (useful for procedure calls).

"offset" is a SIGNED CONSTANT encoded as part of the instruction!

BEQ(ra,offset,rc): Branch if equal

```
NPC ← PC + 4
Reg[rc] ← NPC
if (Reg[ra] == 0)
      PC ← NPC + 4*offset
else
      PC ← NPC
```

BNE(ra,offset,rc): Branch if not equal

```
NPC ← PC + 4
Reg[rc] ← NPC
if (Reg[ra] != 0)
      PC ← NPC + 4*offset
else
      PC ← NPC
```

offset = distance in words to branch target, counting from the instruction following the BEQ/BNE. Range: -32768 to +32767.

Branch instructions also use the instruction format with the 16-bit signed constant. The operation of the branch instructions are a bit complicated, so let's walk through their operation step-by-step.

Let's start by looking at the operation of the BEQ instruction. First the usual PC+4 calculation is performed, giving us the address of the instruction following the BEQ. This value is written to the "rc" register whether or not the branch is taken. This feature of branches is pretty handy and we'll use it to implement procedure calls a couple of lectures from now. Note that if we don't need to remember the PC+4 value, we can specify R31 as the "rc" register.

Next, BEQ tests the value of the "ra" register to see if it's equal to 0. If it is equal to 0, the branch is taken and the PC is incremented by the amount specified in the constant field of the instruction. Actually the constant, called an offset since we're using it to offset the PC, is treated as a word offset and is multiplied by 4 to convert it a byte offset since the PC uses byte addressing. If the contents of the "ra" register is not equal to 0, the PC is incremented by 4 and execution continues with the instruction following the BEQ.

Let me say a few more words about the offset. The branches are using what's referred to as "pc-relative addressing". That means the address of the branch target is specified relative to the address of the branch, or, actually, relative to the address of the instruction following the branch. So an offset of 0 would refer to the instruction following the branch and an offset of -1 would refer to the branch itself. Negative offsets are called "backwards branches" and are usually seen at branches used at the end of loops, where the looping condition is tested and we branch backwards to the beginning of the loop if another iteration is called for. Positive offsets are called "forward branches" and are usually seen in code for "if statements", where we might skip over some part of the program if a condition is not true.

We can use BEQ to implement a so-called unconditional branch, *i.e.*, a branch that is always taken. If we test R31 to see if it's 0, that's always true, so BEQ(R31,…) would always branch to the specified target.

There's also a BNE instruction, identical to BEQ in its operation except the sense of the condition is reversed: the branch is taken if the value of register "ra" is non-zero.

It might seem that only testing for zero/non-zero doesn't let us do everything we might want to do. For example, how would we branch if "a < b"? That's where the compare instructions come in — they do

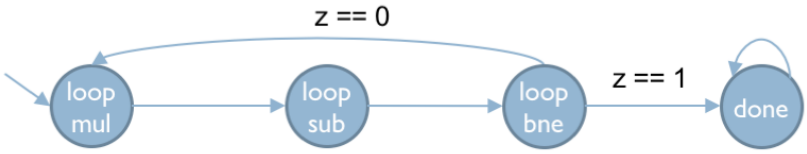
more complicated comparisons, producing a non-zero value if the comparison is true and a zero value if the comparison is false. Then we can use BEQ and BNE to test the result of the comparison and branch appropriately.

Can We Solve Factorial Now?

```
int a = 1;
int b = N;
do {
  a = a * b;
  b = b - 1;
} while (b != 0)
```

```
                        // Assume r1 = N
    ADDC(r31, 1, r0)    // r0 = 1
L:MUL(r0, r1, r0)       // r0 = r0 * r1
    SUBC(r1, 1, r1)     // r1 = r1 - 1
    BNE(r1, L, r31)     // if r1 != 0, run MUL next
                        // at this point, r0 = N!
```

- Remember control FSM for our simple programmable datapath?



- Control FSM states → instructions!
  - Not the case in general
  - Happens here because datapath is similar to basic von Neumann datapath

At long last we're finally in a position to write Beta code to compute factorial using the iterative algorithm shown in C code on the left. In the Beta code, the loop starts at the second instruction and is marked with the "L:" label. The body of the loop consists of the required multiplication and the decrement of b. Then, in the fourth instruction, b is tested and, if it's non-zero, the BNE will branch back to the instruction with the label L.

Note that in our symbolic notation for BEQ and BNE instructions we don't write the offset directly since that would be a pain to calculate and would change if we added or removed instructions from the loop. Instead we reference the instruction to which we want to branch, and the program that translates the symbolic code into the binary instruction fields will do the offset calculation for us.

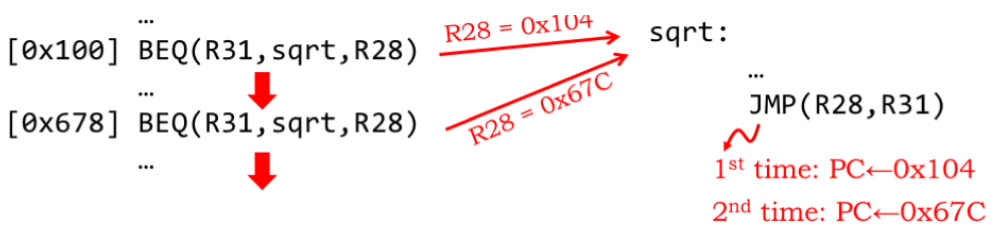
There's a satisfying similarity between the Beta code and the operations specified by the high-level FSM we created for computing factorial in the simple programmable datapath discussed earlier in this lecture. In this example, each state in the high-level FSM matches up nicely with a particular Beta instruction. We wouldn't expect that high degree of correspondence in general, but since our Beta datapath and the example datapath were very similar, the states and instructions match up pretty well.

Beta JMP Instruction

Branches transfer control to some predetermined destination specified by a constant in the instruction. It will be useful to be able to transfer control to a computed address.

JMP(Ra,Rc): Reg[Rc] ← PC + 4
PC ← Reg[Ra]

Useful for procedure call return…



Finally, our last instruction! Branches conditionally transfer control to a specific target instruction. But we'll also need the ability to compute the address of the target instruction — that ability is provided by the JMP instruction which simply sets the program counter to value from register "ra". Like branches, JMP will write the PC+4 value into to the specified destination register.

This capability is very useful for implementing procedures in Beta code. Suppose we have a procedure "sqrt" that computes the square root of its argument, which is passed in, say, R0. We don't show the code for sqrt on the right, except for the last instruction, which is a JMP.

On the left we see that the programmer wants to call the sqrt procedure from two different places in his program. Let's watch what happens…

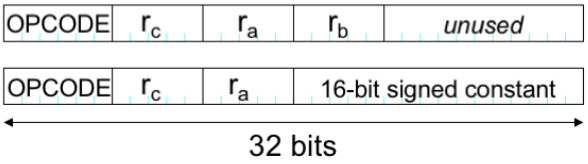
The first call to the sqrt procedure is implemented by the unconditional branch at location 0x100 in main memory. The branch target is the first instruction of the sqrt procedure, so execution continues there. The BEQ also writes the address of the following instruction (0x104) into its destination register, R28. When we reach the end of first procedure call, the JMP instruction loads the value in R28, which is 0x104, into the PC, so execution continues with the instruction following the first BEQ. So we've managed to return from the procedure and continue execution where we left off in the main program.

When we get to the second call to the sqrt procedure, the sequence of events is the same as before except that this time R28 contains 0x67C, the address of the instruction following the second BEQ. So the second time we reach the end of the sqrt procedure, the JMP sets the PC to 0x67C and execution resumes with the instruction following the second procedure call.

Neat! The BEQs and JMP have worked together to implement procedure call and return. We'll discuss the implementation of procedures in detail in an upcoming lecture.

Beta ISA Summary

- Storage:
  - Processor: 32 registers (r31 hardwired to 0) and PC
  - Main memory: 32-bit byte addresses; each memory access involves a 32-bit word. Since there are 4 bytes/word, all addresses will be a multiple of 4.
- Instruction formats:



- Instruction types:
  - ALU: Two input registers, or register and constant
  - Loads and stores
  - Branches, Jumps

That wraps up the design of the Beta instruction set architecture. In summary, the Beta has 32 registers to hold values that can be used as operands for the ALU. All other values, along with the binary representation of the program itself, are stored in main memory. The Beta supports 32-bit

memory addresses and can access values in $2^{32} = 4$ gigabytes of main memory. All Beta memory access refer to 32-bit words, so all addresses will be a multiple of 4 since there are 4 bytes/word.

The are two instruction formats. The first specifies an opcode, two source registers and a destination register. The second replaces the second source register with a 32-bit constant, derived by sign-extending a 16-bit constant stored in the instruction itself.

There are three classes of instructions: ALU operations, LD and ST for accessing main memory, and branches and JMPs that change the order of execution.

And that's it! As we'll see in the next lecture, we'll be able parlay this relatively simple repertoire of operations into a system that can execute any computation we can specify.

- BackDesigning an Instruction Set
- ContinueTopic Videos

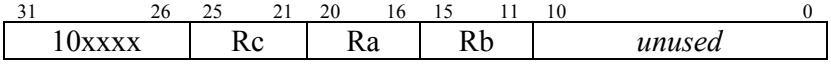
Computation Structures

Instruction Set Architecture Worksheet

Summary of β Instruction Formats

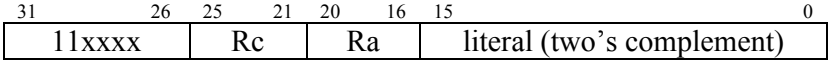
Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

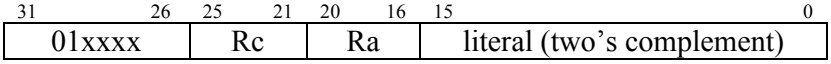
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or), **XNOR** (bitwise exclusive nor),
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or), **XNORC** (bitwise exclusive nor)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/**BF**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/**BT**(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 *5:3*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**	**BEQ**	**BNE**		**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**	**XNOR**	**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**	**XNORC**	**SHLC**	**SHRC**	**SRAC**	

Problem 1.

An unnamed associate of yours has broken into the computer (a Beta of course!) that 6.004 uses for course administration. He has managed to grab the contents of the memory locations he believes holds the Beta code responsible for checking access passwords and would like you to help discover how the password code works. The memory contents are shown in the table below:

	Addr	Contents	Opcode	Rc	Ra	Rb	Assembly
	0x100	0xC05F0008	110000	00010	11111	____	ADDC (R31,8,R2)
	0x104	0xC03F0000	110000	00001	11111	____	ADDC (R31,0,R1)
L2:	0x108	0xE060000F	111000	00011	00000	____	ANDC (R0,15,R3)
	0x10C	0xF0210004	111100	00001	00001	____	SHLC (R1,4,R1)
✓	0x110	0xA4230800	101001	00001	00011	1	OR (R3,R1,R1)
	0x114	0xF4000004	111101	00000	00000	____	SHRC (R0,4,R0)
	0x118	0xC4420001	110001	00010	00010	____	SUBC (R2,1,R2)
	0x11C	0x73E20002	011100	11111	00010	____	BEQ (R2,L1,R31)
☆	0x120	0x73FFFFF9	011100	11111	11111	____	BEQ (R31,L2,R31)
✓	0x124	0xA4230800	101001	00001	00011	1	OR(R3,R1,R1)
L1:	0x128	0x605F0124	011000	00010	11111	____	LD (R31,0x124,R2)
✓	0x12C	0x90211000	100100	00001	00001	2	CMPEQ (R1,R2,R1)

Further investigation reveals that the password is just a 32-bit integer which is in R0 when the code above is executed and that the system will grant access if R1 = 1 after the code has been executed. What "passnumber" will gain entry to the system?

0x8032AA

R0 R1 R2 R3

x 0 8

0 x%16

x%16 x%16

R2 = 8

R1 = 0

R3 = R0 & 15

R1 *= 16

R1 = R1 | R3

R0 /= 16

R2 -= 1

if R2 != 0

nibble

for r2 in 1...8

R2 = R0 의 4비트

R1 = R1 | R3

R1 = R1 | R3

R2 = MEM[292]

R1 = 1 if R1 == R2

A4230800

Problem 2.

(A) What assembly instruction could a compiler use to implement y = x * 8 on the Beta assuming that MUL and MULC are not available? Assume x is in R0 and y is in R1.

Equivalent assembly instruction: SHLC(R0, 3, R1)

(B) Assume that the registers are initialized to: R0=8, R1=10, R2=12, R3=0x1234, R4=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. SHL(R3, R4, R5) **Value of R5:** 0x12340000
2. ADD(R2, R1, R6) **Value of R6:** 22
3. ADD(R0, 2, R7) **Value of R7:** 10
4. ST(R1, 4, R3) **Value stored:** 0x1234 **at address:** 26

(C) A student tries to optimize his Beta assembly program by replacing a line containing

ADDC(R0, 3*4+5, R1)

by

ADDC(R0, 17, R1)

Is the resulting binary program smaller? Does it run faster?

(circle one) Binary program is SMALLER? yes … (no)

(circle one) FASTER? yes … (no)

(D) A BR instruction at location 0x1000 branches to 0x2000. If the binary representation for that BR were moved to location 0x1400 and executed there, where will the relocated instruction branch to?

Branch target for relocated BR (in hex): 0x 2400

(E) A line in an assembly-language program containing "ADDC(R1,2,R3)" is changed to "ADDC(R1,R2,R3)". Will the modified program behave differently when executed?

Circle best answer: YES … (NO) … CAN'T TELL

Problem 3.

Each of the following programs is loaded into a Beta's main memory starting at location 0 and execution is started with the Beta's PC set to 0. Assume that all registers have been initialized to 0 before execution begins. Please determine the specified values after execution reaches the HALT() instruction and the Beta stops. Write "CAN'T TELL" if the value cannot be determined. **Please write all values in hex.**

(A)

```
        . = 0
        LD(R31,X+4,R1)
        SHLC(R1,2,R1)
        LD(R1,X,R2)
        HALT()
X:      LONG(4)
        LONG(3)
        LONG(2)
        LONG(1)
        LONG(0)
```

R1= 12

R2=

Value left in R1: 0x C

Value left in R2: 0x 1

20 1000/0011/0000/1100/0011/1000/0000

(B)

```
        . = 0
0       LD(R31,X,R0)
4       CMOVE(0,R1)
8   L:  CMPLTC(R0,0,R2)
C       BNE(R2,DONE)
10      ADDC(R1,1,R1)
14      SHLC(R0,1,R0)
18      BR(L)
1C  DONE: HALT()
20  X:  LONG(0x08306352)
```

R0= 0x08306352

R0<0 → DONE

R1 += 1

R0 <<1

Value left in R0: 0x 08306352

Value left in R1: 0x 4

Value left in R2: 0x 1

Value assembler assigns to symbol X: 0x 20

(C)

```
        . = 0
0       LD(R31,Z,R1)
4       SHRC(R1,26,R1)
8Z:     CMPLTC(R1,0x3C,R2)
        HALT()
```

R1=

R1= 0

Value left in R1: 0x 0

Value left in R2: 0x 1

(D)

```
        . = 0
0       LD(R31,X,R0)
4       CMOVE(0,R1)
8   L:  ADDC(R1,1,R1)
C       SHRC(R0,1,R0)
10      BNE(R0,L,R2)
14      HALT()

        . = 0x100
0X100 X: LONG(5)
```

101

R0= 5

R1=0

R1+=1

R0 >> 1

if R0!=0

R2=0X14

Value left in R0: 0x 0

Value left in R1: 0x 3

Value left in R2: 0x 14

Value assembler assigns to symbol X: 0x 53

252

(256-4)/4= 83

=0X53

1C

r0 = 0x87654321

r1 = 12

r2 = 17

(E)

```
        . = 0
        LD(r31, X, r0)
        CMPLE(r0, r31, r1)
        BNE(r1, L1, r1)
        ADDC(r31, 17, r2)
        BEQ(r31, L2, r31)
L1:     SRAC(r0, 4, r2)
L2:     HALT()

        . = 0x1CE8
X:      LONG(0x87654321)
```

Value left in R0? 0x 87654321

Value left in R1? 0x C

Value left in R2? 0x 15

Value assembler assigns to L1: 0x 2

(F)

```
        . = 0
        LD(R31, i, R0)
        SHLC(R0, 2, R0)
        LD(R0, a-4, R1)
        HALT()
a:      LONG(0xBADBABE)
        LONG(0xDEADBEEF)
        LONG(0xC0FFEE)
        LONG(0x8BADF00D)
i:      LONG(3)
```

R0 = 3

R0 = 12

R1 = 0xC0FFEE

Contents of R0 (in hex): 0x C

Contents of R1 (in hex): 0x C0FFEE

R1 = F443003C

R1 = 0x1111 0000 1000 0011 003C

R2 = F443

(G)

```
        . = 0
        LD(R31,Z,R1)
        SHRC(R1,16,R2)
Z:      SUBC(R2,0x3C,R3)
        HALT()
```

- 3C

F406

Value left in R1: 0x F443003C

Value left in R3: 0x F406

Value assembler assigns to symbol Z: 0x 8

(H)

```
0       . = 0
4       LD(R31,X,R0)
        CMOVE(0,R1)
8   L:  ADDC(R1,1,R1)
12      SHRC(R0,1,R0)
16      BNE(R0,L,R2)
20      HALT()

        X:  LONG(0xDECAF)
```

1101

R0 = 0x DECAF

R1 = 0

R1 += 1

R0 >> 1

if R0 ≠ 0

R2 = 0x14

Value left in R0: 0x 0

Value left in R1: 0x 14

Value left in R2: 0x 14

Problem 1.

An unnamed associate of yours has broken into the computer (a Beta of course!) that 6.004 uses for course administration. He has managed to grab the contents of the memory locations he believes holds the Beta code responsible for checking access passwords and would like you to help discover how the password code works. The memory contents are shown in the table below:

	Addr	Contents	Opcode	Rc	Ra	Rb	Assembly
	0x100	0xC05F0008	110000	00010	11111	_____	ADDC(R31, 0x8, R2)
	0x104	0xC03F0000	110000	00001	11111	_____	ADDC(R31, 0x0, R1)
L2:	0x108	0xE060000F	111000	00011	00000	_____	ANDC(R0, 0xF, R3)
	0x10C	0xF0210004	111100	00001	00001	_____	SHLC(R1, 0x4, R1)
	0x110	0xA4230800	101001	00001	00011	00001	OR(R3, R1, R1)
	0x114	0xF4000004	111101	00000	00000	_____	SHRC(R0, 0x4, R0)
	0x118	0xC4420001	110001	00010	00010	_____	SUBC(R2, 0x1, R2)
	0x11C	0x73E20002 (+2)	011100	11111	00010	_____	BEQ(R2, L1, R31)
	0x120	0x73FFFFF9 (-7)	011100	11111	11111	_____	BEQ(R31, L2, R31)
	0x124	0xA4230800	101001	00001	00011	_____	not executed!
L1:	0x128	0x605F0124	011000	00010	11111	_____	LD(R31, 0x124, R2)
	0x12C	0x90211000	100100	00001	00001	_____	CMPEQ(R1, R2, R1)

Further investigation reveals that the password is just a 32-bit integer which is in R0 when the code above is executed and that the system will grant access if R1 = 1 after the code has been executed. What "passnumber" will gain entry to the system?

The loop reverses the order of the nibbles (4-bit chunks) of the value in R0, eg., 0x12345678 becomes 0x87654321.

So the "passnumber" is the nibble reverse of 0xA4230800 which is 0x00830324A.

Problem 2.

(A) What assembly instruction could a compiler use to implement y = x * 8 on the Beta assuming that MUL and MULC are not available? Assume x is in R0 and y is in R1.

Equivalent assembly instruction: SHLC(R0,3,R1)

(B) Assume that the registers are initialized to: R0=8, R1=10, R2=12, R3=0x1234, R4=24 before execution of each of the following assembly instructions. For each instruction, provide the value of the specified register or memory location. **If your answers are in hexadecimal, make sure to prepend them with the prefix 0x.**

1. SHL(R3, R4, R5) **Value of R5:** 0x3400 0000
2. ADD(R2, R1, R6) **Value of R6:** 22
3. ADD(R0, 2, R7) **Value of R7:** 20
4. ST(R1, 4, R3) **Value stored:** 10 **at address:** 4 + 0x1234 = 0x1238

op = ADDC so interpreted as R2!

(C) A student tries to optimize his Beta assembly program by replacing a line containing

ADDC(R0, 3*4+5, R1)

by

ADDC(R0, 17, R1)

expression value computed at assembly time

Is the resulting binary program smaller? Does it run faster?

(circle one) Binary program is SMALLER? yes ... **no**

(circle one) FASTER? yes ... **no**

(D) A BR instruction at location 0x1000 branches to 0x2000. If the binary representation for that BR were moved to location 0x1400 and executed there, where will the relocated instruction branch to?

Original branch offset (0x1000) now relative to 0x1400

Branch target for relocated BR (in hex): 0x 2400

(E) A line in an assembly-language program containing "ADDC(R1,2,R3)" is changed to "ADDC(R1,R2,R3)". Will the modified program behave differently when executed?

Interpret 2nd operand as a constant expression.

Value of symbol R2 is 2.

Circle best answer: YES ... NO ... CAN'T TELL

Problem 3

Each of the following programs is loaded into a Beta's main memory starting at location 0 and execution is started with the Beta's PC set to 0. Assume that all registers have been initialized to 0 before execution begins. Please determine the specified values after execution reaches the HALT() instruction and the Beta stops. Write "CAN'T TELL" if the value cannot be determined. **Please write all values in hex.**

(A)

```
        . = 0
        LD(R31,X+4,R1)
        SHLC(R1,2,R1)
        LD(R1,X,R2)
        HALT()
   X:   LONG(4)
   +4   LONG(3)
   +8   LONG(2)
   +12  LONG(1)
        LONG(0)
```

R1 ← 3
R1 ← 12
R2 ← Mem[x+12]

Value left in R1: 0x C

Value left in R2: 0x 1

(B)

```
0       LD(R31,X,R0)
4       CMOVE(0,R1)
8  L:   CMPLTC(R0,0,R2)
C       BNE(R2,DONE)
10      ADDC(R1,1,R1)
14      SHLC(R0,1,R0)
18      BR(L)
1C DONE: HALT()
20 X:   LONG(0x08306352)
```

Value left in R0: 0x 83063520

Value left in R1: 0x 4

Value left in R2: 0x 1

Value assembler assigns to symbol X: 0x 20

↑ counts # of left shifts needed until MSB of R0 is 1.

(C)

```
        . = 0
        LD(R31,Z,R1)
        SHRC(R1,26,R1)
   Z:   CMPLTC(R1,0x3C,R2)
        HALT()
```

R1 ← binary for CMPLTC inst.
R1 ← opcode field

Value left in R1: 0x 35 (CMPLTC opcode)

Value left in R2: 0x 1

(D)

```
0       LD(R31,X,R0)
4       CMOVE(0,R1)
8  L:   ADDC(R1,1,R1)
C       SHRC(R0,1,R0)
10      BNE(R0,L,R2)
14      HALT()

        . = 0x100
   X:   LONG(5)
```

R0 ← 5

Value left in R0: 0x 0

Value left in R1: 0x 3

Value left in R2: 0x 14

Value assembler assigns to symbol X: 0x 100

↑ count # of right shifts until R0 == 0.

(E)

```
        . = 0
0       LD(r31, X, r0)
4       CMPLE(r0, r31, r1)
8       BNE(r1, L1, r1)
C       ADDC(r31, 17, r2)
10      BEQ(r31, L2, r31)
14  L1: SRAC(r0, 4, r2)
18  L2: HALT()

        . = 0x1CE8
    X:  LONG(0x87654321)
```

R0 ← 0x87654321 (negative!)

R1 ← 1

R1 ← 0xC, branch taken

R2 ← 0xF8765432

Value left in R0? 0x 87654321

Value left in R1? 0x C

Value left in R2? 0x F8765432

Value assembler assigns to L1: 0x 14

(F)

```
        . = 0
        LD(R31, i, R0)
        SHLC(R0, 2, R0)
        LD(R0, a-4, R1)
        HALT()
    a:  LONG(0xBADBABE)
+4      LONG(0xDEADBEEF)
+8      LONG(0xC0FFEE)
        LONG(0x8BADF00D)
    i:  LONG(3)
```

R0 ← 3

R0 ← 12

R1 ← Mem[12 + a - 4]

= i + 8

Contents of R0 (in hex): 0x C

Contents of R1 (in hex): 0x C0FFEE

(G)

```
        . = 0
0       LD(R31,Z,R1)
4       SHRC(R1,16,R2)
8   Z:  SUBC(R2,0x3C,R3)
C       HALT()
```

R1 ← binary for SUBC

R2 ← top half R1

SUBC 3 2

110001	00011	00010	0x3C
op	R2	R3	

Value left in R1: 0x C4620003C

Value left in R3: 0x C426

Value assembler assigns to symbol Z: 0x 8

(H)

```
        . = 0
0       LD(R31,X,R0)
4       CMOVE(0,R1)

8   L:  ADDC(R1,1,R1)
C       SHRC(R0,1,R0)
10      BNE(R0,L,R2)
14      HALT()

    X:  LONG(0xDECAF)
```

R0 ← DECAF

↑ count # of right shifts to make R0 equal to zero.

Value left in R0: 0x 0

Value left in R1: 0x 14

Value left in R2: 0x 14

10 Assembly Language, Models of Computation

10.1 ANNOTATED SLIDES

← BROWSE COURSE MATERIAL

- « Assembly Language, Models of Computation
- 10.1.1Annotated slides
- » Topic Videos

.toc { margin-left: 2em; } .lecslide { margin-top: 1em; margin-bottom: 1em; border-top: 0.5px solid #808080; padding-top: 1em; text-align: center; } .lecslideimg { width: 5in; border: 1px solid black; }

L10a: Assembly Language

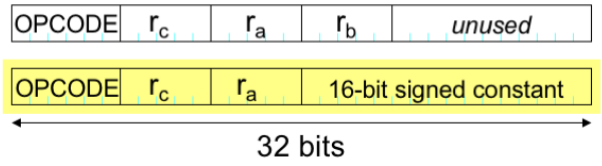
Content of the following slides is described in the surrounding text.

Beta ISA Summary

- Storage:

- Processor: 32 registers (r31 hardwired to 0) and PC
- Main memory: Up to 4 GB, 32-bit words, 32-bit byte addresses, 4-byte-aligned accesses

- Instruction formats:



- Instruction classes:
  - ALU: Two input registers, or register and constant
  - Loads and stores: access memory
  - Branches, Jumps: change program counter

In the previous lecture we developed the instruction set architecture for the Beta, the computer system we'll be building throughout this part of the course. The Beta incorporates two types of storage or memory. In the CPU datapath there are 32 general-purpose registers, which can be read to supply source operands for the ALU or written with the ALU result. In the CPU's control logic there is a special-purpose register called the program counter, which contains the address of the memory location holding the next instruction to be executed.

The datapath and control logic are connected to a large main memory with a maximum capacity of 2^{32} bytes, organized as 2^{30} 32-bit words. This memory holds both data and instructions.

Beta instructions are 32-bit values comprised of various fields. The 6-bit OPCODE field specifies the operation to be performed. The 5-bit Ra, Rb, and Rc fields contain register numbers, specifying one of the 32 general-purpose registers. There are two instruction formats: one specifying an opcode and three registers, the other specifying an opcode, two registers, and a 16-bit signed constant.

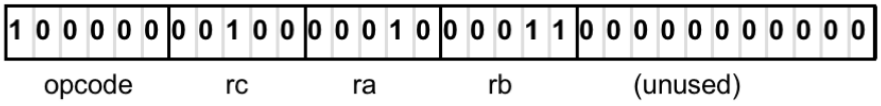
There three classes of instructions. The ALU instructions perform an arithmetic or logic operation on two operands, producing a result that is stored in the destination register. The operands are either two values from the general-purpose registers, or one register value and a constant. The yellow highlighting indicates instructions that use the second instruction format.

The Load/Store instructions access main memory, either loading a value from main memory into a register, or storing a register value to main memory.

And, finally, there are branches and jumps whose execution may change the program counter and hence the address of the next instruction to be executed.

Programming Languages

32-bit (4-byte) ADD instruction:



Means, to the BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write in *assembly language*:

```
ADD(R2, R3, R4)
```

Today

or better yet a *high-level language*:

```
a = b + c;
```

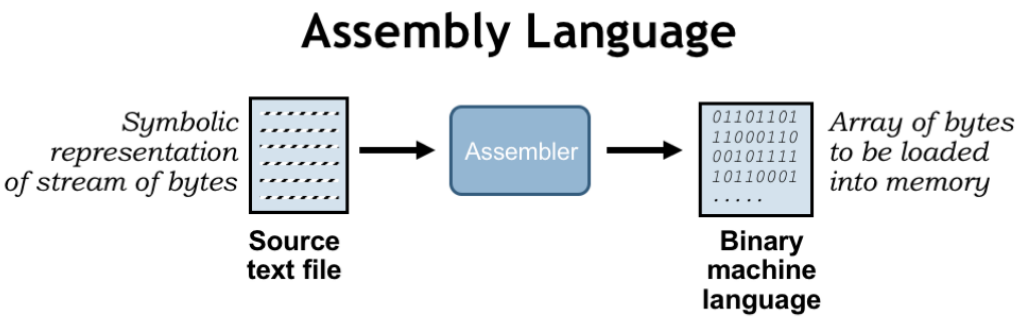
Coming up

To program the Beta we'll need to load main memory with binary-encoded instructions. Figuring out each encoding is clearly the job for a computer, so we'll create a simple programming language that will let us specify the opcode and operands for each instruction. So instead of writing the binary at the top of slide, we'll write assembly language statements to specify instructions in symbolic form. Of course we still have think about which registers to use for which values and write sequences of instructions for more complex operations.

By using a high-level language we can move up one more level abstraction and describe the computation we want in terms of variables and mathematical operations rather than registers and ALU functions.

In this lecture we'll describe the assembly language we'll use for programming the Beta. And in the next lecture we'll figure out how to translate high-level languages, such as C, into assembly language.

The layer cake of abstractions gets taller yet: we could write an interpreter for say, Python, in C and then write our application programs in Python. Nowadays, programmers often choose the programming language that's most suitable for expressing their computations, then, after perhaps many layers of translation, come up with a sequence of instructions that the Beta can actually execute.



- Abstracts bit-level representation of instructions and addresses
- We'll learn UASM ("microassembler"), built into BSim
- Main elements:
  - Values
  - Symbols
  - Labels (symbols for addresses)
  - Macros

Okay, back to assembly language, which we'll use to shield ourselves from the bit-level representations of instructions and from having to know the exact location of variables and instructions in memory. A program called the "assembler" reads a text file containing the assembly language program and produces an array of 32-bit words that can be used to initialize main memory.

We'll learn the UASM assembly language, which is built into BSim, our simulator for the Beta ISA. UASM is really just a fancy calculator! It reads arithmetic expressions and evaluates them to produce 8-bit values, which it then adds sequentially to the array of bytes which will eventually be loaded into the Beta's memory. UASM supports several useful language features that make it easier to write assembly language programs. Symbols and labels let us give names to particular values and addresses. And macros let us create shorthand notations for sequences of expressions that, when

evaluated, will generate the binary representations for instructions and data.

Example UASM Source File

```
        N = 12              // loop index initial value
        ADDC(r31, N, r1)    // r1 = loop index
        ADDC(r31, 1, r0)    // r0 = accumulated product
loop:   MUL(r0, r1, r0)     // r0 = r0 * r1
        SUBC(r1, 1, r1)     /* r1 = r1 - 1 */
        BNE(r1, loop, r31)  // if r1 != 0, NextPC=loop
```

- Comments after //, ignored by assembler (also /*…*/)
- Symbols are symbolic representations of a constant value (they are NOT variables!)
- Labels are symbols for addresses
- Macros expand into sequences of bytes
  - Most frequently, macros are instructions
  - We can use them for other purposes

Here's an example UASM source file. Typically we write one UASM statement on each line and can use spaces, tabs and newlines to make the source as readable as possible. We've added some color coding to help in our explanation.

Comments (shown in green) allow us to add text annotations to the program. Good comments will help remind you how your program works. You really don't want to have figure out from scratch what a section of code does each time you need to modify or debug it! There are two ways to add comments to the code. "//" starts a comment, which then occupies the rest of the source line. Any characters after "//" are ignored by the assembler, which will start processing statements again at the start of the next line in the source file. You can also enclose comment text using the delimiters "/*" and "*/" and the assembler will ignore everything in-between. Using this second type of comment, you can "comment-out" many lines of code by placing "/*" at the start and, many lines later, end the comment section with "*/".

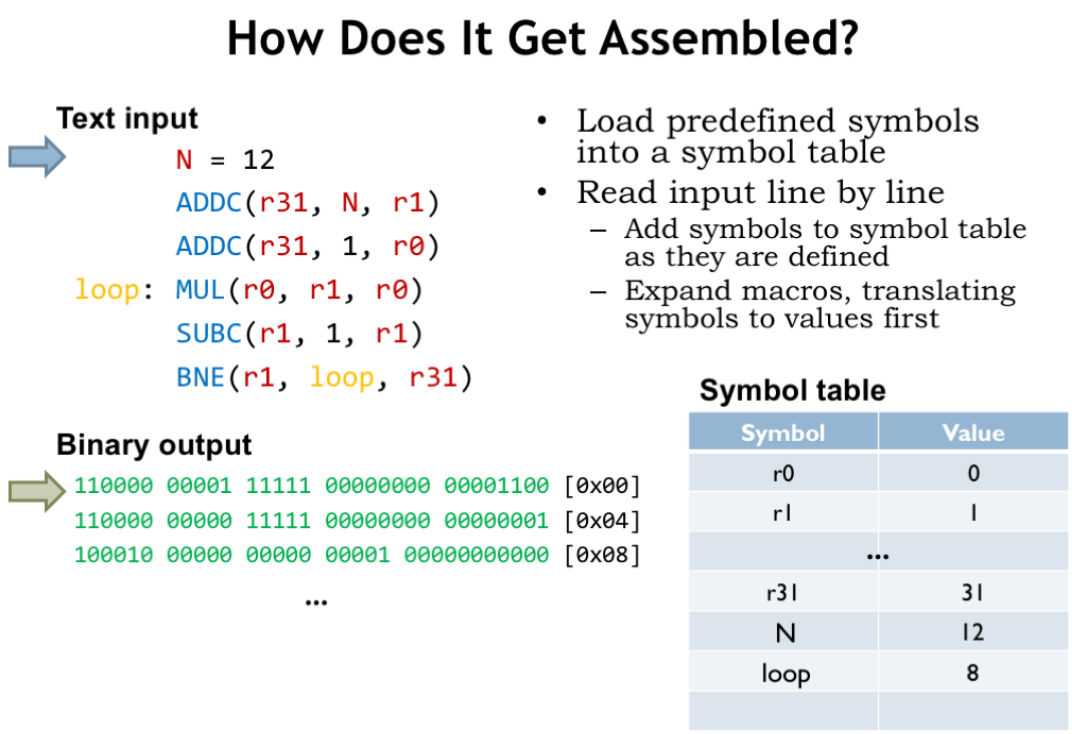
Symbols (shown in red) are symbolic names for constant values. Symbols make the code easier to understand, *e.g.*, we can use N as the name for an initial value for some computation, in this case the value 12. Subsequent statements can refer to this value using the symbol N instead of entering the value 12 directly. When reading the program, we'll know that N means this particular initial value. So if later we want to change the initial value, we only have to change the definition of the symbol N rather than find all the 12's in our program and change them. In fact some of the other appearances of 12 might not refer to this initial value and so to be sure we only changed the ones that did, we'd have to read and understand the whole program to make sure we only edited the right 12's. You can imagine how error-prone that might be! So using symbols is a practice you want to follow!

Note that all the register names are shown in red. We'll define the symbols R0 through R31 to have the values 0 through 31. Then we'll use those symbols to help us understand which instruction operands are intended to be registers, *e.g.*, by writing R1, and which operands are numeric values, *e.g.*, by writing the number 1. We could just use numbers everywhere, but the code would be much harder to read and understand.

A label (shown in yellow) is a symbol whose value are the address of a particular location in the program. Here, the label "loop" will be our name for the location of the MUL instruction in memory. In the BNE at the end of the code, we use the label "loop" to specify the MUL instruction as the branch target. So if R1 is non-zero, we want to branch back to the MUL instruction and start another iteration.

We'll use indentation for most UASM statements to make it easy to spot the labels defined by the program. Indentation isn't required, it's just another habit assembly language programmers use to keep their programs readable.

We use macro invocations (shown in blue) when we want to write Beta instructions. When the assembler encounters a macro, it "expands" the macro, replacing it with a string of text provided by in the macro's definition. During expansion, the provided arguments are textually inserted into the expanded text at locations specified in the macro definition. Think of a macro as shorthand for a longer text string we could have typed in. We'll see how all this works in the next video segment.



Let's follow along as the assembler processes our source file. The assembler maintains a symbol table that maps symbols' names to their numeric values. Initially the symbol table is loaded with mappings for all the register symbols.

The assembler reads the source file line-by-line, defining symbols and labels, expanding macros, or evaluating expressions to generate bytes for the output array. Whenever the assembler encounters a use of a symbol or label, it's replaced by the corresponding numeric value found in the symbol table.

The first line, N = 12, defines the value of the symbol N to be 12, so the appropriate entry is made in the symbol table.

Advancing to the next line, the assembler encounters an invocation of the ADDC macro with the arguments "r31", "N", and "r1". As we'll see in a couple of slides, this triggers a series of nested macro expansions that eventually lead to generating a 32-bit binary value to be placed in memory location 0. The 32-bit value is formatted here to show the instruction fields and the destination address is shown in brackets.

The next instruction is processed in the same way, generating a second 32-bit word.

On the fourth line, the label loop is defined to have the value of the location in memory that's about to filled (in this case, location 8). So the appropriate entry is made in the symbol table and the MUL macro is expanded into the 32-bit word to be placed in location 8.

The assembler processes the file line-by-line until it reaches the end of the file. Actually the assembler makes two passes through the file. On the first pass it loads the symbol table with the values from all the symbol and label definitions. Then, on the second pass, it generates the binary output. The two-pass approach allows a statement to refer to symbol or label that is defined later in the file, *e.g.*, a forward branch instruction could refer to the label for an instruction later in the program.

Registers are Predefined Symbols

- r0 = 0, …, r31 = 31
- Treated like normal symbols:

```
ADDC(r31, N, r1)
        ↓  Substitute symbols with their values
ADDC(31, 12, 1)
        ↓  Expand macro
110000 00001 11111 00000000 00001100
```

- No “type checking” if you use the wrong opcode…

```
ADDC(r31, r12, r1)          ADD(r31, N, r1)
        ↓                           ↓
ADDC(31, 12, 1)             ADD(31, 12, 1)
Reg[1] ← Reg[31] + 12       Reg[1] ← Reg[31] + Reg[12]
```

As we saw in the previous slide, there’s nothing magic about the register symbols — they are just symbolic names for the values 0 through 31. So when processing ADDC(r31,N,r1), UASM replaces the symbols with their values and actually expands ADDC(31,12,1).

UASM is very simple. It simply replaces symbols with their values, expands macros and evaluates expressions. So if you use a register symbol where a numeric value is expected, the value of the symbol is used as the numeric constant. Probably not what the programmer intended.

Similarly, if you use a symbol or expression where a register number is expected, the low-order 5 bits of the value is used as the register number, in this example, as the Rb register number. Again probably not what the programmer intended.

The moral of the story is that when writing UASM assembly language programs, you have to keep your wits about you and recognize that the interpretation of an operand is determined by the opcode macro, not by the way you wrote the operand.

Labels and Offsets

Input file

```
        N = 12
        ADDC(r31, N, r1)
        ADDC(r31, 1, r0)
loop:   MUL(r0, r1, r0)
        SUBC(r1, 1, r1)
        BNE(r1, loop, r31)
```

Output file

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00001 00000 00000000000 [0x08]
110001 00001 00001 00000000 00000001 [0x0C]
011101 11111 00001 11111111 11111101 [0x10]
```

```
offset = (label - <addr of BNE/BEQ>)/4 - 1
```

- Label value is the address of a memory location
- BEQ/BNE macros compute offset automatically
- Labels hide addresses!

Symbol table

Symbol	Value
r0	0
r1	1
…	
r31	31
N	12
loop	8

```
= (8 - 16)/4 - 1 = -3
```

Recall from Lecture 9 that branch instructions use the 16-bit constant field of the instruction to encode the address of the branch target as a word offset from the location of the branch instruction. Well, actually the offset is calculated from the instruction immediately following the branch, so an offset of -1 would refer to the branch itself.

The calculation of the offset is a bit tedious to do by hand and would, of course, change if we added or removed instructions between the branch instruction and branch target. Happily macros for the branch instructions incorporate the necessary formula to compute the offset from the address of the branch and the address of the branch target. So we just specify the address of the branch target, usually with a label, and let UASM do the heavy lifting.

Here we see that BNE branches backwards by three instructions (remember to count from the instruction following the branch) so the offset is -3. The 16-bit two's complement representation of -3 is the value placed in the constant field of the BNE instruction.

Mighty Macroinstructions

Macros are parameterized abbreviations, or shorthand

```
// Macro to generate 4 consecutive bytes:
.macro consec(n)  n  n+1  n+2  n+3

// Invocation of above macro:
consec(37)
```

Is expanded to

```
⇒ 37 37+1 37+2 37+3 ⇒ 37  38  39  40
```

Here are macros for breaking multi-byte data types into byte-sized chunks

```
// Assemble into bytes, little-endian:
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x >> 16)

. = 0x100
LONG(0xdeadbeef)
```

Has same effect as:

```
      0xef   0xbe   0xad   0xde
Mem: 0x100  0x101 0x102  0x103
```

Boy, that's hard to read. Maybe, those big-endian types do have a point.

Let's take a closer look at how macros work in UASM. Here we see the definition of the macro "consec" which has a single parameter "n". The body of the macro is a sequence of four expressions. When there's an invocation of the "consec" macro, in this example with the argument 37, the body of the macro is expanded replacing all occurrences of "n" with the argument 37. The resulting text is then processed as if it had appeared in place of the macro invocation. In this example, the four expressions are evaluated to give a sequence of four values that will be placed in the next four bytes of the output array.

Macro expansions may contain other macro invocations, which themselves will be expanded, continuing until all that's left are expressions to be evaluated. Here we see the macro definition for WORD, which assembles its argument into two consecutive bytes. And for the macro LONG, which assembles its argument into four consecutive bytes, using the WORD macro to process the low 16 bits of the value, then the high 16 bits of the value.

These two UASM statements cause the constant 0xDEADBEEF to converted to 4 bytes, which are deposited in the output array starting at index 0x100.

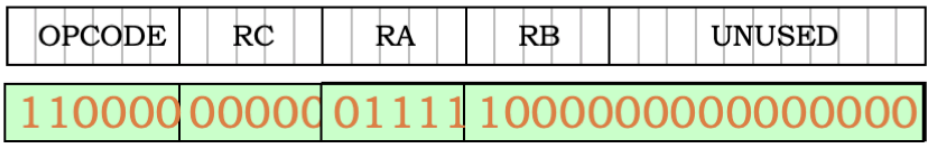
Note that the Beta expects the least-significant byte of a multi-byte value to be stored at the lowest byte address. So the least-significant byte 0xEF is placed at address 0x100 and the most-significant byte 0xDE is placed at address 0x103. This is the "little-endian" convention for multi-byte values: the

least-significant byte comes first. Intel's x86 architecture is also little-endian.

There is a symmetrical "big-endian" convention where the most-significant byte comes first. Both conventions are in active use and, in fact, some ISAs can be configured to use either convention! There's no "right answer" for which convention to use, but the fact that there two conventions means that we have to be alert for the need to convert the representation of multi-byte values when moving values between one ISA and another, *e.g.*, when we send a data file to another user.

As you can imagine there are strong advocates for both schemes who are happy to defend their point of view at great length. Given the heat of the discussion, it's appropriate that the names for the conventions were drawn from Jonathan Swift's "Gulliver's Travels" in which a civil war is fought over whether to open a soft-boiled egg at its big end or its little end.

Assembly of Instructions




```
// Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

// Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
    .align 4
    LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC % 0x10000))
}

// Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL) betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

".align 4" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

For example:

```
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)

ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```

Let's look at the macros used to assemble Beta instructions. The BETAOP helper macro supports the 3-register instruction format, taking as arguments the values to be placed in the OPCODE, Ra, Rb, and Rc fields. The ".align 4" directive is a bit of administrative bookkeeping to ensure that instructions will have a byte address that's a multiple of 4, *i.e.*, that they span exactly one 32-bit word in memory. That's followed by an invocation of the LONG macro to generate the 4 bytes of binary data representing the value of the expression shown here. The expression is where the actual assembly of the fields takes place. Each field is limited to requisite number of bits using the modulo operator (%), then shifted left (<<) to the correct position in the 32-bit word.

And here are the helper macros for the instructions that use a 16-bit constant as the second operand.

Let's follow the assembly of an ADDC instruction to see how this works. The ADDC macro expands into an invocation of the BETAOPC helper macro, passing along the correct value for the ADDC opcode, along with the three operands.

The BETAOPC macro does the following arithmetic: The OP argument, in this case the value 0x30, is shifted left to occupy the high-order 6 bits of the instruction. Then the RA argument, in this case 15, is placed in its proper location. The 16-bit constant -32768 is positioned in the low 16 bits of the instruction. And, finally, the Rc argument, in this case 0, is positioned in the Rc field of the instruction.

You can see why we call this processing "assembling an instruction". The binary representation of an instruction is assembled from the binary values for each of the instruction fields. It's not a complicated process, but it requires a lot of shifting and masking, tasks that we're happy to let a computer handle.

Example Assembly

```
ADDC(R3,1234,R17)
```
expand ADDC macro with RA=R3, C=1234, RC=R17
```
betaopc(0x30,R3,1234,R17)
```
expand betaopc macro with OP=0x30, RA=R3, CC=1234, RC=R17
```
.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%32)<<16)+(1234 % 0x10000))
```
expand LONG macro with X=0xC22304D2
```
WORD(0xC22304D2)    WORD(0xC22304D2 >> 16)
```
expand first WORD macro with X=0xC22304D2
```
0xC22304D2%256    (0xC22304D2/256)%256    WORD(0xC223)
```
evaluate expressions, expand second WORD macro with X=0xC223
```
0xD2    0x04    0xC223%256    (0xC223/256)%256
```
evaluate expressions
```
0xD2    0x04    0x23    0xC2
```

Here's the entire sequence of macro expansions that assemble this ADDC instruction into an appropriate 32-bit binary value in main memory.

You can see that the knowledge of Beta instruction formats and opcode values is built into the bodies of the macro definitions. The UASM processing is actually quite general — with a different set of macro definitions it could process assembly language programs for almost any ISA!

UASM Macros for Beta Instructions

(defined in beta.uasm)

```
| BETA Instructions:
.macro ADD(RA,RB,RC)   betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)   betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)   betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)   betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)   betaopc(0x32,RA,C,RC)
   :
   :
.macro LD(RA,CC,RC)    betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)   betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)    betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)   betaopc(0x19,R31,CC,RC)
   :
   :
.macro BEQ(RA,LABEL,RC) betabr(0x1C,RA,RC,LABEL)
.macro BEQ(RA,LABEL)   betabr(0x1C,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)
.macro BNE(RA,LABEL)   betabr(0x1D,RA,r31,LABEL)
```

Convenience macros so we don't have to specify R31…

All the macro definitions for the Beta ISA are provided in the beta.uasm file, which is included in each of the assembly language lab assignments. Note that we include some convenience macros to define shorthand representations that provide common default values for certain operands. For example, except for procedure calls, we don't care about the PC+4 value saved in the destination register by branch instructions, so almost always would specify R31 as the Rc register, effectively discarding the PC+4 value saved by branches. So we define two-argument branch macros that automatically provide R31 as the destination register. Saves some typing, and, more importantly, it makes it easier to

understand the assembly language program.

Pseudoinstructions

- Convenience macros that expand to one or more real instructions
- Extend set of operations without adding instructions to the ISA

```
// Convenience macros so we don't have to use R31
.macro LD(CC,RC)        LD(R31,CC,RC)
.macro ST(RA,CC)        ST(RA,CC,R31)
.macro BEQ(RA,LABEL)    BEQ(RA,LABEL,R31)
.macro BNE(RA,LABEL)    BNE(RA,LABEL,R31)

.macro MOVE(RA,RC)          ADD(RA,R31,RC)      // Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)         ADDC(R31,C,RC)      // Reg[RC] <- C
.macro COM(RA,RC)           XORC(RA,-1,RC)      // Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)           SUB(R31,RB,RC)      // Reg[RC] <- -Reg[RB]
.macro NOP()                ADD(R31,R31,R31)    // do nothing

.macro BR(LABEL)            BEQ(R31,LABEL)      // always branch
.macro BR(LABEL,RC)         BEQ(R31,LABEL,RC)   // always branch
.macro CALL(LABEL)          BEQ(R31,LABEL,LP)   // call subroutine
.macro BF(RA,LABEL,RC)      BEQ(RA,LABEL,RC)    // 0 is false
.macro BF(RA,LABEL)         BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)      BNE(RA,LABEL,RC)    // 1 is true
.macro BT(RA,LABEL)         BNE(RA,LABEL)

// Multi-instruction sequences
.macro PUSH(RA)             ADDC(SP,4,SP)   ST(RA,-4,SP)
.macro POP(RA)              LD(SP,-4,RA)    ADDC(SP,-4,SP)
```

Here are a whole set of convenience macros intended to make programs more readable. For example, unconditional branches can be written using the BR() macro rather than the more cumbersome BEQ(R31,…). And it's more readable to use branch-false (BF) and branch-true (BT) macros when testing the results of a compare instruction.

And note the PUSH and POP macros at the bottom of page. These expand into multi-instruction sequences, in this case to add and remove values from a stack data structure pointed to by the SP register.

We call these macros "pseudo instructions" since they let us provide the programmer with what appears a larger instruction set, although underneath the covers we've just using the same small instruction repertoire developed in Lecture 9.

Factorial with Pseudoinstructions

Before

```
      N = 12
      ADDC(r31, N, r1)
      ADDC(r31, 1, r0)
loop: MUL(r0, r1, r0)
      SUBC(r1, 1, r1)
      BNE(r1, loop, r31)
```

After

```
      N = 12
      CMOVE(N, r1)
      CMOVE(1, r0)
loop: MUL(r0, r1, r0)
      SUBC(r1, 1, r1)
      BNE(r1, loop)
```

In this example we've rewritten the original code we had for the factorial computation using pseudo instructions. For example, CMOVE is a pseudo instruction for moving small constants into a register. It's easier for us to read and understand the intent of a "constant move" operation than an "add a value to 0" operation provided by the ADDC expansion of CMOVE. Anything we can do to remove the cognitive clutter will be very beneficial in the long run.

Raw Data

- LONG assembles a 32-bit value
  - Variables
  - Constants > 16 bits

```
N:     LONG(12)
factN: LONG(0xdeadbeef)
       …
Start:
       LD(N, r1)
       CMOVE(1, r0)
loop:  MUL(r0, r1, r0)
       SUBC(r1, 1, r1)
       BT(r1, loop)
       ST(r0, factN)
```

Symbol table

Symbol	Value
…	
N	0
factN	4

```
LD(r31, N, r1)
LD(31, 0, 1)

Reg[1] ← Mem[Reg[31] + 0]
       ← Mem[0]
       ← 12
```

So far we've talked about assembling instructions. What about data? How do we allocate and initialize data storage and how do we get those values into registers so that they can be used as operands?

Here we see a program that allocates and initializes two memory locations using the LONG macro. We've used labels to remember the addresses of these locations for later reference.

When the program is assembled the values of the label N and factN are 0 and 4 respectively, the addresses of the memory locations holding the two data values.

To access the first data value, the program uses a LD instruction, in this case one of convenience macros that supplies R31 as the default value of the Ra field. The assembler replaces the reference to the label N with its value 0 from the symbol table. When the LD is executed, it computes the memory address by adding the constant (0) to the value of the Ra register (which is R31 and hence the value is 0) to get the address (0) of the memory location from which to fetch the value to be placed in R1.

UASM Expressions and Layout

- Values can be written as expressions
  - Assembler evaluates expressions, they are *not* translated to instructions to compute the value!

```
A = 7 + 3 * 0x0cc41
B = A - 3
```

- The "." (period) symbol means the next byte address to be filled
  - Can read or write to it
  - Useful to control data layout or leave empty space (e.g., for

arrays)

```
. = 0x100          // Assemble into 0x100
LONG(0xdeadbeef)
k = .              // Symbol "k" has value 0x104
LONG(0x00dec0de)
. = .+16           // Skip 16 bytes
LONG(0xc0ffeeee)
```

The constants needed as values for data words and instruction fields can be written as expressions. These expressions are evaluated by the assembler as it assembles the program and the resulting value is used as needed. Note that the expressions are evaluated at the time the assembler runs. By the time the program runs on the Beta, the resulting value is used. The assembler does NOT generate ADD and MUL instructions to compute the value during program execution. If a value is needed for an instruction field or initial data value, the assembler has to be able to perform the arithmetic itself. If you need the program to compute a value during execution, you have to write the necessary instructions as part of your program.

One last UASM feature: there's a special symbol ".", called "dot", whose value is the address of the next main memory location to be filled by the assembler when it generates binary data. Initially "." is 0 and it's incremented each time a new byte value is generated.

We can set the value of "." to tell the assembler where in memory we wish to place a value. In this example, the constant 0xDEADBEEF is placed into location 0x100 of main memory. And we can use "." in expressions to compute the values for other symbols, as shown here when defining the value for the symbol "k". In fact, the label definition "k:" is exactly equivalent to the UASM statement "k = ."

We can even increment the value of "." to skip over locations, *e.g.*, if we wanted to leave space for an un initialized array.

Summary: Assembly Language

- Low-level language, symbolic representation of sequence of bytes. Abstracts:
  - Bit-level representation of instructions
  - Addresses
- Elements: Values, symbols, labels, macros
- Values can be constants or expressions
- Symbols are symbolic representations of values
- Labels are symbols for addresses
- Macros are expanded to byte sequences:
  - Instructions
  - Pseudoinstructions (translate to 1+ real instructions)
  - Raw data
- Can control where to assemble with "." symbol

And that's assembly language! We use assembly language as a convenient notation for generating the binary encoding for instructions and data. We let the assembler build the bit-level representations we need and to keep track of the addresses where these values are stored in main memory.

UASM itself provides support for values, symbols, labels and macros.

Values can be written as constants or expressions involving constants.

We use symbols to give meaningful names to values so that our programs will be more readable and

We use symbols to give meaningful names to values so that our programs will be more readable and more easily modified. Similarly, we use labels to give meaningful names to addresses in main memory and then use the labels in referring to data locations in LD or ST instructions, or to instruction locations in branch instructions.

Macros hide the details of how instructions are assembled from their component fields.

And we can use "." to control where the assembler places values in main memory.

The assembler is itself a program that runs on our computer. That raises an interesting "chicken and egg problem": how did the first assembler program get assembled into binary so it could run on a computer? Well, it was hand-assembled into binary. I suspect it processed a very simple language indeed, with the bells and whistles of symbols, labels, macros, expression evaluation, etc. added only after basic instructions could be assembled by the program. And I'm sure they were very careful not to lose the binary so they wouldn't have to do the hand-assembly a second time!

Universality?

- Recall: We say a set of Boolean gates is universal if we can implement any Boolean function using only gates from that set.
- What problems can we solve with a von Neumann computer? (e.g., the Beta)
  - Everything that FSMs can solve?
  - Every problem?
  - Does it depend on the ISA?
- Needed: a mathematical model of computation
  - Prove what can be computed, what can't

An interesting question for computer architects is what capabilities must be included in the ISA? When we studied Boolean gates in Part 1 of the course, we were able to prove that NAND were universal, *i.e.*, that we could implement any Boolean function using only circuits constructed from NAND gates.

We can ask the corresponding question of our ISA: is it universal, *i.e.*, can it be used to perform any computation? what problems can we solve with a von Neumann computer? Can the Beta solve any problem FSMs can solve? Are there problems FSMs can't solve? If so, can the Beta solve those problems? Do the answers to these questions depend on the particular ISA?

To provide some answers, we need a mathematical model of computation. Reasoning about the model, we should be able to prove what can be computed and what can't. And hopefully we can ensure that the Beta ISA has the functionality needed to perform any computation.

Models of Computation

The roots of computer science stem from the evaluation of many alternative

- switches

mathematical "models" of computation to determine the classes of computations each could represent.

An elusive goal was to find a universal model, capable of representing *all* practical computations...

- gates
- combinational logic
- memories
- FSMs

Are FSMs the ultimate digital computing device?

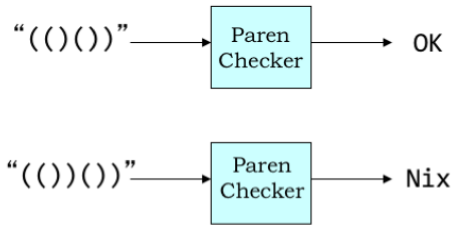
The roots of computer science stem from the evaluation of many alternative mathematical models of computation to determine the classes of computation each could represent. An elusive goal was to find a universal model, capable of representing *all* realizable computations. In other words if a computation could be described using some other well-formed model, we should also be able to describe the same computation using the universal model.

One candidate model might be finite state machines (FSMs), which can be built using sequential logic. Using Boolean logic and state transition diagrams we can reason about how an FSM will operate on any given input, predicting the output with 100% certainty.

Are FSMs the universal digital computing device? In other words, can we come up with FSM implementations that implement all computations that can be solved by any digital device?

FSM Limitations

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For instance:



Well-formed Parentheses Checker:

Given any string of coded left & right parens, outputs 1 if it is balanced, else 0.

Simple, easy to describe.

Can this problem be solved using an FSM??? NO!

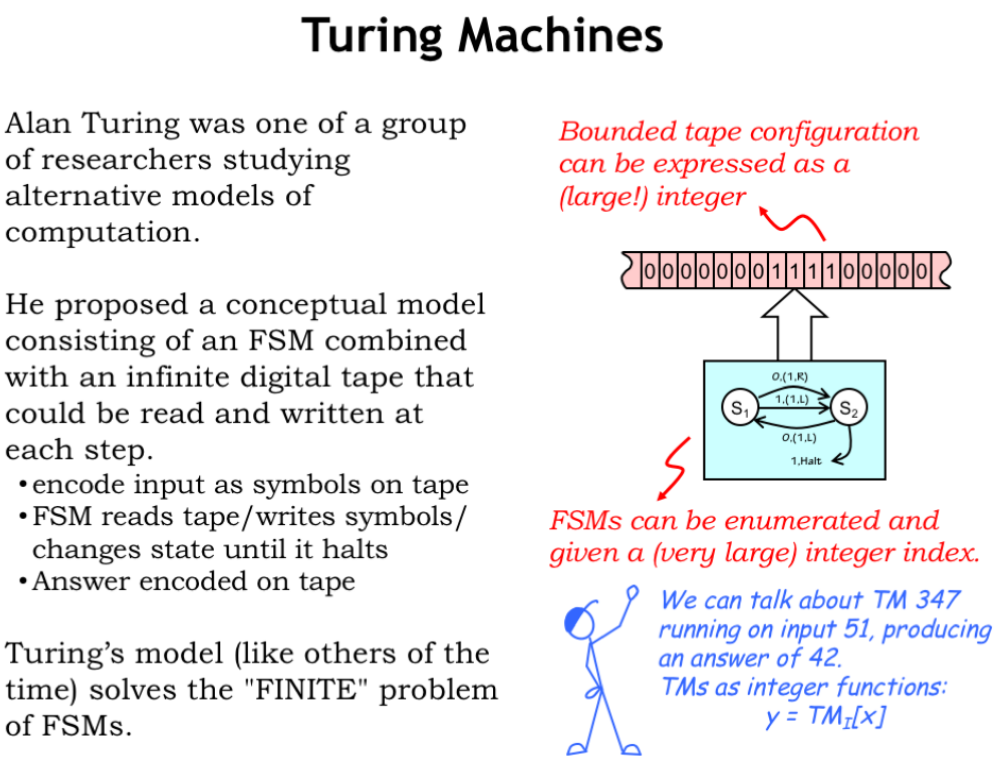
PROBLEM: Requires *arbitrarily* many states, depending on input. Must "COUNT" unmatched left parens. An FSM can only keep track of a finite number of unmatched parens: for every FSM, we can find a string it can't check.

Alan Turing

Despite their usefulness and flexibility, there are common problems that cannot be solved by any FSM. For example, can we build an FSM to determine if a string of parentheses (properly encoded into a binary sequence) is well-formed? A parenthesis string is well-formed if the parentheses balance, *i.e.*, for every open parenthesis there is a matching close parenthesis later in the string. In the example shown here, the input string on the top is well-formed, but the input string on the bottom is not. After processing the input string, the FSM would output a 1 if the string is well-formed, 0 otherwise.

Can this problem be solved using an FSM? No, it can't. The difficulty is that the FSM uses its internal state to encode what it knows about the history of the inputs. In the paren checker, the FSM would need to count the number of unbalanced open parens seen so far, so it can determine if future input contains the required number of close parens. But in a finite state machine there are only a fixed number of states, so a particular FSM has a maximum count it can reach. If we feed the FSM an input with more open parens than it has the states to count, it won't be able to check if the input string is well-formed.

The "finite-ness" of FSMs limits their ability to solve problems that require unbounded counting. Hmm, what other models of computation might we consider? Mathematics to the rescue, in this case in the form of a British mathematician named Alan Turing.



In the early 1930's Alan Turing was one of many mathematicians studying the limits of proof and computation. He proposed a conceptual model consisting of an FSM combined with a infinite digital tape that could read and written under the control of the FSM. The inputs to some computation would be encoded as symbols on the tape, then the FSM would read the tape, changing its state as it performed the computation, then write the answer onto the tape and finally halting. Nowadays, this model is called a Turing Machine (TM). Turing Machines, like other models of the time, solved the "finite" problem of FSMs.

So how does all this relate to computation? Assuming the non-blank input on the tape occupies a finite number of adjacent cells, it can be expressed as a large integer. Just construct a binary number using the bit encoding of the symbols from the tape, alternating between symbols to the left of the tape head and symbols to the right of the tape head. Eventually all the symbols will be incorporated into the (very large) integer representation.

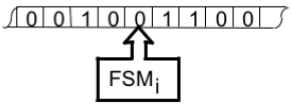
So both the input and output of the TM can be thought of as large integers, and the TM itself as implementing an integer function that maps input integers to output integers.

The FSM brain of the Turing Machine can be characterized by its truth table. And we can systematically enumerate all the possible FSM truth tables, assigning an index to each truth table as it appears in the enumeration. Note that indices get very large very quickly since they essentially incorporate all the information in the truth table. Fortunately we have a very large supply of integers!

We'll use the index for a TM's FSM to identify the TM as well. So we can talk about TM 347 running on input 51, producing the answer 42.

Other Models of Computation…

Turing Machines [Turing]



Alan Turing

Recursive Functions [Kleene]

$F(0,x) \equiv x$

$F(1+y,x) \equiv 1+F(x,y)$

```
(define (fact n)
  (... (fact (- n 1))
```

Stephen Kleene

Lambda calculus [Church, Curry, Rosser…]

Alonzo Church

$\lambda x.\lambda y.xxy$

```
(lambda(x)(lambda(y)(x (x y))))
```

Production Systems [Post, Markov]

Emile Post

$\alpha \rightarrow \beta$

```
IF pulse=0 THEN
   patient=dead
```

There are many other models of computation, each of which describes a class of integer functions where a computation is performed on an integer input to produce an integer answer. Kleene, Post and Turing were all students of Alonzo Church at Princeton University in the mid-1930's. They explored many other formulations for modeling computation: recursive functions, rule-based systems for string rewriting, and the lambda calculus. They were all particularly intrigued with proving the existence of problems unsolvable by realizable machines. Which, of course, meant characterizing the problems that could be solved by realizable machines.

Computability

FACT: Each model studied is capable of computing exactly the same set of integer functions!

Proof Technique:
Constructions that translate between models

BIG IDEA:
Computability, independent of computation scheme chosen

Church's Thesis:

Every discrete function computable by ANY realizable machine is computable by some Turing machine.

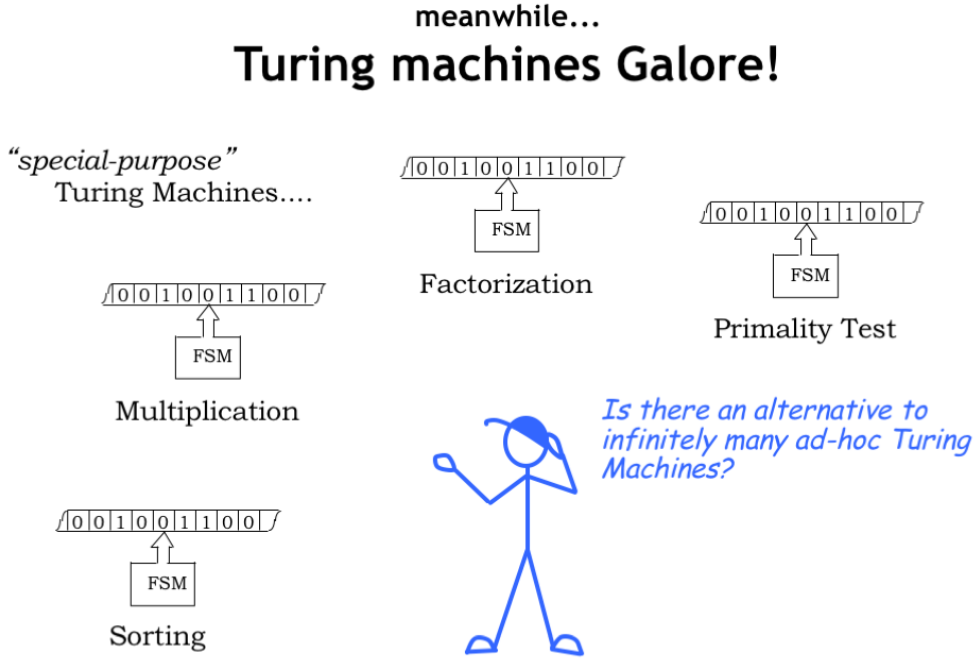
f(x) *computable* ⇔ for some k, all x
$f(x) = T_k[x]$

It turned out that each model was capable of computing *exactly* the same set of integer functions! This was proved by coming up with constructions that translated the steps in a computation between the various models. It was possible to show that if a computation could be described by one model, an equivalent description exists in the other model. This lead to a notion of computability that was independent of the computation scheme chosen. This notion is formalized by Church's Thesis, which says that every discrete function computable by any realizable machine is computable by some

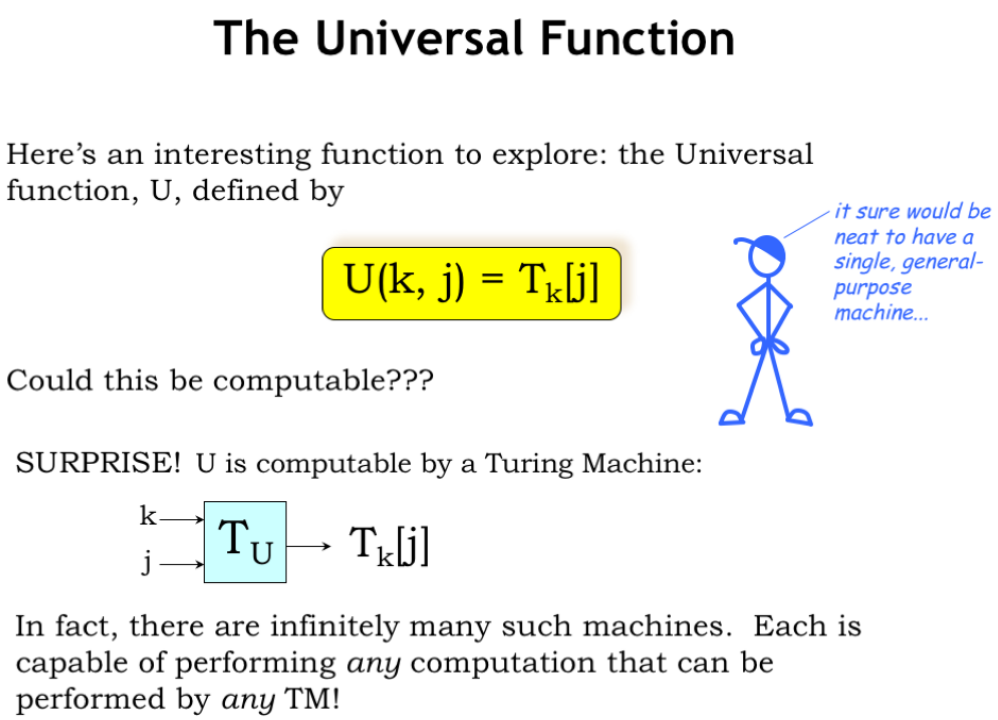
says that every discrete function computable by any realizable machine is computable by some Turing Machine. So if we say the function f(x) is computable, that's equivalent to saying that there's a TM that given x as an input on its tape will write f(x) as an output on the tape and halt.

As yet there's no proof of Church's Thesis, but it's universally accepted that it's true. In general "computable" is taken to mean "computable by some TM".

If you're curious about the existence of uncomputable functions, please see the optional video at the end of this lecture.



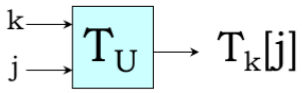
Okay, we've decided that Turing Machines can model any realizable computation. In other words for every computation we want to perform, there's a (different) Turing Machine that will do the job. But how does this help us design a general-purpose computer? Or are there some computations that will require a special-purpose machine no matter what?



What we'd like to find is a universal function U: it would take two arguments, k and j, and then compute the result of running T_k on input j. Is U computable, *i.e.*, is there a universal Turing Machine T_U? If so, then instead of many ad-hoc TMs, we could just use T_U to compute the results for any computable function.

Surprise! U is computable and T_U exists. If fact there are infinitely many universal TMs, some quite simple - the smallest known universal TM has 4 states and uses 6 tape symbols. A universal machine is capable of performing any computation that can be performed by any TM!

Universality



What's going on here?

k encodes a "program" – a description of some arbitrary machine.

j encodes the input data to be used.

T_U *interprets* the program, emulating its processing of the data!

KEY IDEA: Interpretation.
Manipulate *coded representations* of computing machines, rather than the machines themselves.

What's going on here? k encodes a "program" - a description of some arbitrary TM that performs a particular computation. j encodes the input data on which to perform that computation. T_U "interprets" the program, emulating the steps T_k will take to process the input and write out the answer. The notion of interpreting a coded representation of a computation is a key idea and forms the basis for our stored program computer.

Turing Universality

The *Universal Turing Machine* is the paradigm for modern general-purpose computers!

Basic threshold test: Is your computer *Turing Universal*?

- If so, it can emulate every other Turing machine!
- Thus, your computer can compute any computable function

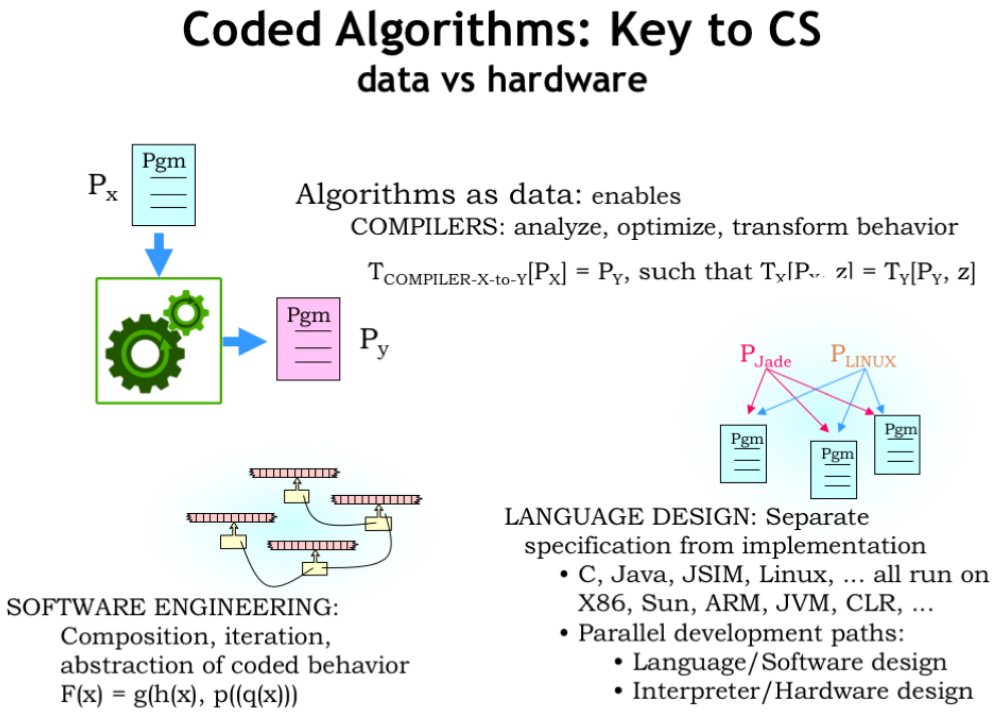
To show your computer is Universal: demonstrate that it can emulate some known UTM.

- Actually given finite memory, can only emulate UTMs + inputs up to a certain size
- This is not a high bar: conditional branches (BEQ) and some simple arithmetic (SUB) are enough.

The Universal Turing Machine is the paradigm for modern general-purpose computers. Given an ISA we want to know if it's equivalent to a universal Turing Machine. If so, it can emulate every other TM and hence compute any computable function.

How do we show our computer is Turing Universal? Simply demonstrate that it can emulate some known Universal Turing Machine. The finite memory on actual computers will mean we can only emulate UTM operations on inputs up to a certain size, but within this limitation we can show our computer can perform any computation that fits into memory.

As it turns out this is not a high bar: so long as the ISA has conditional branches and some simple arithmetic, it will be Turing Universal.



This notion of encoding a program in a way that allows it to be data to some other program is a key idea in computer science.

We often translate a program Px written to run on some abstract high-level machine (eg, a program in C or Java) into, say, an assembly language program Py that can be interpreted by our CPU. This translation is called compilation.

Much of software engineering is based on the idea of taking a program and using it as as component in some larger program.

Given a strategy for compiling programs, that opens the door to designing new programming languages that let us express our desired computation using data structures and operations particularly suited to the task at hand.

So what have we learned from the mathematicians' work on models of computation? Well, it's nice to know that the computing engine we're planning to build will be able to perform any computation that can be performed on any realizable machine. And the development of the universal Turing Machine model paved the way for modern stored-program computers. The bottom line: we're good to go with the Beta ISA!

Uncomputable functions: There are well-defined discrete functions that a Turing machine cannot compute

- No algorithm can compute f(x) for arbitrary x in finite number of steps
- Not that we don't know algorithm - can prove no algorithm exists
- Corollary: Finite memory is not the only limiting factor on whether we can solve a problem

The most famous uncomputable function is the so-called Halting function, $f_H(k, j)$, defined by:

$$f_H(k, j) = \begin{cases} 1 & \text{if } T_k[j] \text{ halts;} \\ 0 & \text{otherwise.} \end{cases}$$

$f_H(k, j)$ determines whether the k^{th} TM halts when given a tape containing j.

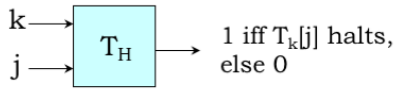
We've discussed computable functions. Are there uncomputable functions?

Yes, there are well-defined discrete functions that cannot be computed by any TM, *i.e.*, no algorithm can compute f(x) for arbitrary finite x in a finite number of steps. It's not that we don't know the algorithm, we can actually prove that no algorithm exists. So the finite memory limitation of FSMs wasn't the only barrier as to whether we can solve a problem.

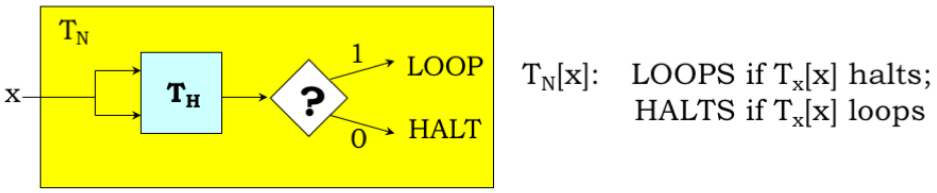
The most famous uncomputable function is the so-called Halting function. When TMs undertake a computation there two possible outcomes. Either the TM writes an answer onto the tape and halts, or the TM loops forever. The Halting function tells which outcome we'll get: given two integer arguments k and j, the Halting function determines if the kth TM halts when given a tape containing j as the input.

Why f_H is Uncomputable

If f_H is computable, it is equivalent to some TM (say, T_H):



Then T_N (N for "Nasty"), which must be computable if T_H is:



$T_N[x]$: LOOPS if $T_x[x]$ halts; HALTS if $T_x[x]$ loops

Finally, consider giving N as an argument to T_N:

$T_N[N]$: LOOPS if $T_N[N]$ halts; HALTS if $T_N[N]$ loops

T_N can't be computable, hence T_H can't either!

Let's quickly sketch an argument as to why the Halting function is not computable. Well, suppose it was computable, then it would be equivalent to some TM, say T_H.

So we can use T_H to build another TM, T_N (the "N" stands for nasty!) that processes its single argument and either LOOPs or HALTs. $T_N[X]$ is designed to loop if TM X given input X halts. And vice versa: $T_N[X]$ halts if TM X given input X loops. The idea is that $T_N[X]$ does the opposite of whatever $T_X[X]$ does. T_N is easy to implement assuming that we have T_H to answer the "halts or

whatever $T_X[X]$ does. T_N is easy to implement assuming that we have T_H to answer the "halts or loops" question.

Now consider what happens if we give N as the argument to T_N. From the definition of T_N, $T_N[N]$ will LOOP if the halting function tells us that $T_N[N]$ halts. And $T_N[N]$ will HALT if the halting function tells us that $T_N[N]$ loops. Obviously $T_N[N]$ can't both LOOP and HALT at the same time! So if the Halting function is computable and T_H exists, we arrive at this impossible behavior for $T_N[N]$. This tells us that T_H cannot exist and hence that the Halting function is not computable.

- BackAssembly Language, Models of Computation
- ContinueTopic Videos

11 Compilers

11.1 ANNOTATED SLIDES

← BROWSE COURSE MATERIAL

- « Compilers
- 11.1.1Annotated slides
- » Topic Videos

.toc { margin-left: 2em; } .lecslide { margin-top: 1em; margin-bottom: 1em; border-top: 0.5px solid #808080; padding-top: 1em; text-align: center; } .lecslideimg { width: 5in; border: 1px solid black; }

L11: Compilers

Content of the following slides is described in the surrounding text.

Programming Languages

COURSE INFO

Instructor:

Course Number:

Departments:

As Taught In:

Level:

TOPICS

Engineering

Computer Sci

Computer D

Electrical Enc

Digital Syste

LEARNING RESOU

Lecture Videos

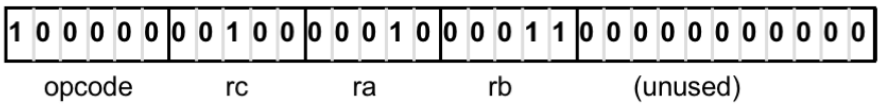
Programming

Lecture Notes

Feedback

Programming Languages

32-bit (4-byte) ADD instruction:



Means, to BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write

`ADD(R2, R3, R4)` ***(Assembly)***

or better yet

`a = b + c;` ***(High-Level Language)***

Today we're going to talk about how to translate high-level languages into code that computers can execute.

So far we've seen the Beta ISA, which includes instructions that control the datapath operations performed on 32-bit data stored in the registers. There are also instructions for accessing main memory and changing the program counter. The instructions are formatted as opcode, source, and destination fields that form 32-bit values in main memory.

To make our lives easier, we developed assembly language as a way of specifying sequences of instructions. Each assembly language statement corresponds to a single instruction. As assembly language programmers, we're responsible for managing which values are in registers and which are in main memory, and we need to figure out how to break down complicated operations, *e.g.*, accessing an element of an array, into the right sequence of Beta operations.

We can go one step further and use high-level languages to describe the computations we want to perform. These languages use variables and other data structures to abstract away the details of storage allocation and the movement of data to and from main memory. We can just refer to a data object by name and let the language processor handle the details. Similarly, we'll write expressions and other operators such as assignment (=) to efficiently describe what would require many statements in assembly language.

Today we're going to dive into how to translate high-level language programs into code that will run on the Beta.

High-Level Languages

Most algorithms are naturally expressed at a high level. Consider the following algorithm:

```
/* Compute greatest common divisor
 * using Euclid's method
 */
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
```

- 6.004 uses C, a common systems programming language. Modern popular alternatives include C++, Java, Python, and many others

- Advantages over assembly
  - Productivity (concise, readable, maintainable)
  - Correctness (type checking, etc)
  - Portability (run same program on different hardware)

```
    y = y - x;
  }
 }
 return x;
}
```

- Disadvantages over assembly?
  - Efficiency?

Implementations: Interpretation vs compilation

Here we see Euclid's algorithm for determining the greatest common divisor of two numbers, in this case the algorithm is written in the C programming language. We'll be using a simple subset of C as our example high-level language. Please see the brief overview of C in the Handouts section if you'd like an introduction to C syntax and semantics. C was developed by Dennis Ritchie at AT&T Bell Labs in the late 60's and early 70's to use when implementing the Unix operating system. Since that time many new high-level languages have been introduced providing modern abstractions like object-oriented programming along with useful new data and control structures.

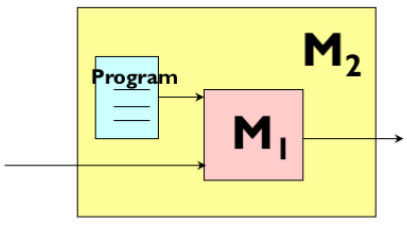
Using C allows us describe a computation without referring to any of the details of the Beta ISA like registers, specific Beta instructions, and so on. The absence of such details means there is less work required to create the program and makes it easier for others to read and understand the algorithm implemented by the program.

There are many advantages to using a high-level language. They enable programmers to be very productive since the programs are concise and readable. These attributes also make it easy to maintain the code. Often it is harder to make certain types of mistakes since the language allows us to check for silly errors like storing a string value into a numeric variable. And more complicated tasks like dynamically allocating and deallocating storage can be completely automated. The result is that it can take much less time to create a correct program in a high-level language than it would it when writing in assembly language.

Since the high-level language has abstracted away the details of a particular ISA, the programs are portable in the sense that we can expect to run the same code on different ISAs without having to rewrite the code.

What do we lose by using a high-level language? Should we worry that we'll pay a price in terms of the efficiency and performance we might get by crafting each instruction by hand? The answer depends on how we choose to run high-level language programs. The two basic execution strategies are "interpretation" and "compilation".

Interpretation

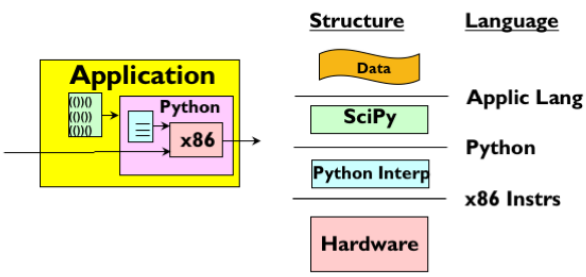


Model of *Interpretation:*

- Start with some hard-to-program machine, say M_1
- Write a single program for M_1 that mimics the behavior of some easier machine, M_2
- Result: a "virtual" M_2

Layers of interpretation:

- Often we use several layers of interpretation to achieve desired behavior, e.g.:
- x86 CPU, running
  - Python, running
    - SciPy application, performing
      - Numerical analysys



To interpret a high-level language program, we'll write a special program called an "interpreter" that runs on the actual computer, M1. The interpreter mimics the behavior of some abstract easy-to-program machine M2 and for each M2 operation executes sequences of M1 instructions to achieve the desired result. We can think of the interpreter along with M1 as an implementation of M2, *i.e.*, given a program written for M2, the interpreter will, step-by-step, emulate the effect of M2 instructions.

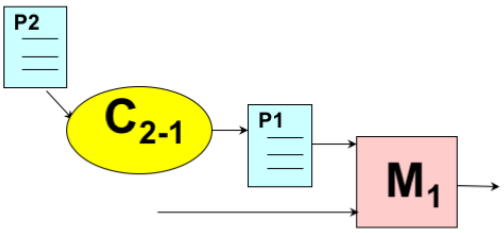
We often use several layers of interpretation when tackling computation tasks. For example, an engineer may use her laptop with an Intel CPU to run the Python interpreter. In Python, she loads the SciPy toolkit, which provides a calculator-like interface for numerical analysis for matrices of data. For each SciPy command, *e.g.*, "find the maximum value of a dataset", the SciPy tool kit executes many Python statements, *e.g.*, to loop over each element of the array, remembering the largest value. For each Python statement, the Python interpreter executes many x86 instructions, *e.g.*, to increment the loop index and check for loop termination. Executing a single SciPy command may require executing of tens of Python statements, which in turn each may require executing hundreds of x86 instructions. The engineer is very happy she didn't have to write each of those instructions herself!

Interpretation is an effective implementation strategy when performing a computation once, or when exploring which computational approach is most effective before making a more substantial investment in creating a more efficient implementation.

Compilation

Model of *Compilation:*

- Given some hard-to-program machine, say M_1…
- Find some easier-to-program language L_2 (perhaps for a more complicated machine, M_2); write programs in that language



- Build a translator (compiler) that translates programs from M_2's language to M_1's language. May run on M_1, M_2, or some other machine.

Interpretation and compilation: two ways to execute high-level languages

- Both allow changes in the source program
- Both afford programming applications in platform (e.g., processor) independent languages
- Both are widely used in modern computer systems!

We'll use a compilation implementation strategy when we have computational tasks that we need to execute repeatedly and hence we are willing to invest more time up-front for more efficiency in the long-term.

In compilation, we also start with our actual computer M1. Then we'll take our high-level language program P2 and translate it statement-by-statement into a program for M1. Note that we're not actually running the P2 program. Instead we're using it as a template to create an equivalent P1 program that can execute directly on M1. The translation process is called "compilation" and the program that does the translation is called a "compiler".

We compile the P2 program once to get the translation P1, and then we'll run P1 on M1 whenever we want to execute P2. Running P1 avoids the overhead of having to process the P2 source and the costs of executing any intervening layers of interpretation. Instead of dynamically figuring out the necessary machine instructions for each P2 statement as it's encountered, in effect we've arranged to capture that stream of machine instructions and save them as a P1 program for later execution. If we're willing to pay the up-front costs of compilation, we'll get more efficient execution.

And, with different compilers, we can arrange to run P2 on many different machines — M2, M3, etc. — without having rewrite P2.

So we now have two ways to execute a high-level language program: interpretation and compilation. Both allow us to change the original source program. Both allow us to abstract away the details of the actual computer we'll use to run the program. And both strategies are widely used in modern computer systems!

Interpretation vs Compilation

- Characteristic differences:

	Interpretation	Compilation
How it treats input "x+2"	Computes x+2	Generates a program that computes x+2
When it happens	During execution	Before execution
What it complicates/slows	Program execution	Program development
Decisions made at	Run time	Compile time

- Major choice we'll see repeatedly: do it at compile time or at run time?
  - Which is faster?
  - Which is more general?

Let's summarize the differences between interpretation and compilation.

Suppose the statement "x+2" appears in the high-level program. When the interpreter processes this statement it immediately fetches the value of the variable x and adds 2 to it. On the other hand, the compiler would generate Beta instructions that would LD the variable x into a register and then ADD 2 to that value.

The interpreter is executing each statement as it's processed and, in fact, may process and execute the same statement many times if, *e.g.*, it was in a loop. The compiler is just generating instructions to be executed at some later time.

Interpreters have the overhead of processing the high-level source code during execution and that overhead may be incurred many times in loops. Compilers incur the processing overhead once, making the eventual execution more efficient. But during development, the programmer may have to compile and run the program many times, often incurring the cost of compilation for only a single execution of the program. So the compile-run-debug loop can take more time.

The interpreter is making decisions about the data type of x and the type of operations necessary at run time, *i.e.*, while the program is running. The compiler is making those decisions during the compilation process.

Which is the better approach? In general, executing compiled code is much faster than running the code interpretively. But since the interpreter is making decisions at run time, it can change its behavior depending, say, on the type of the data in the variable X, offering considerable flexibility in handling different types of data with the same algorithm. Compilers take away that flexibility in exchange for fast execution.

Compilers

- Bare minimum for a functional compiler:

- Good compilers:
  - Produce meaningful errors on incorrect programs
    - Even better: meaningful warnings
  - Produce fast, optimized code

- This lecture:
  - Simple techniques to compile a C programs into assembly
  - Overview of how modern compilers work

A compiler is a program that translates a high-level language program into a functionally equivalent sequence of machine instructions, *i.e.*, an assembly language program.

A compiler first checks that the high-level program is correct, *i.e.*, that the statements are well formed, the programmer isn't asking for nonsensical computations — *e.g.*, adding a string value and an integer — or attempting to use the value of a variable before it has been properly initialized. The compiler may also provide warnings when operations may not produce the expected results, *e.g.*, when converting from a floating-point number to an integer, where the floating-point value may be too large to fit in the number of bits provided by the integer.

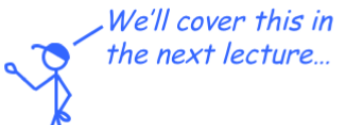
If the program passes scrutiny, the compiler then proceeds to generate efficient sequences of instructions, often finding ways to rearrange the computation so that the resulting sequences are shorter and faster. It's hard to beat a modern optimizing compiler at producing assembly language, since the compiler will patiently explore alternatives and deduce properties of the program that may not be apparent to even diligent assembly language programmers.

In this section, we'll look at a simple technique for compiling C programs into assembly. Then, in the next section, we'll dive more deeply into how a modern compiler works.

A Simple Compilation Strategy

- Programs are sequences of statements, so repeatedly call `compile_statement(`*`statement`*`)`:
  - Unconditional: *`expr`*`;`
  - Compound: `{` *`statement`*$_1$`;` *`statement`*$_2$`;` `...` `}`
  - Conditional: `if (`*`expr`*`)` *`statement`*$_1$`; else` *`statement`*$_2$`;`
  - Iteration: `while (`*`expr`*`)` *`statement`*`;`

- Also need `compile_expr(expr)` to generate code to compute value of expr into a register
  - Constants: `1234`
  - Variables: `a`, `b[`*`expr`*`]`
  - Assignment: `a =` *`expr`*
  - Operations: *`expr`*$_1$ `+` *`expr`*$_2$, …
  - Procedure calls: *`proc(expr,...)`*



There are two main routines in our simple compiler: compile_statement and compile_expr. The job of compile_statement is to compile a single statement from the source program. Since the source program is a sequence of statements, we'll be calling compile_statement repeatedly.

We'll focus on the compilation technique for four types of statements. An unconditional statement is simply an expression that's evaluated once. A compound statement is simply a sequence of statements to be executed in turn. Conditional statements, sometimes called "if statements", compute the value of an test expression, *e.g.*, a comparison such as "A < B". If the test is true then statement_1 is executed, otherwise statement_2 is executed. Iteration statements also contain a test expression. In each iteration, if the test is true, then the statement is executed, and the process repeats. If the test is false, the iteration is terminated.

The other main routine is compile_expr whose job it is to generate code to compute the value of an expression, leaving the result in some register. Expressions take many forms:

simple constant values,

values from scalar or array variables,

assignment expressions that compute a value and then store the result in some variable,

unary or binary operations that combine the values of their operands with the specified operator. Complex arithmetic expressions can be decomposed into sequences of unary and binary operations.

And, finally, procedure calls, where a named sequence of statements will be executed with the values of the supplied arguments assigned as the values for the formal parameters of the procedure. Compiling procedures and procedure calls is a topic that we'll tackle next lecture since there are some complications to understand and deal with.

Happily, compiling the other types of expressions and statements is straightforward, so let's get started.

compile_expr(expr) ⇒ Rx

- Constants: 1234 ⇒ Rx
  - `CMOVE(1234,Rx)`
  - `LD(c1,Rx)`

    …

    `c1: LONG(123456)`
- Variables: a ⇒ Rx
  - `LD(a,Rx)`

    `...`

    `a: LONG(0)`
- Assignment: a=expr ⇒ Rx
  - `compile_expr(expr)⇒Rx`

    `ST(Rx,a)`
- Variables: b[*expr*] ⇒ Rx
  - `compile_expr(expr)⇒Rx`

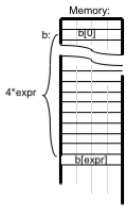
    `MULC(Rx,bsize,Rx)`

    `LD(Rx,b,Rx)`

    …

    `// reserve array space`

    `b: . = . + bsize*blen`



- Operations:

  $expr_1 + expr_2 \Rightarrow Rx$
  - `compile_expr(`$expr_1$`)⇒Rx`

    `compile_expr(`$expr_2$`)⇒Ry`

    `ADD(Rx,Ry,Rx)`

What code do we need to put the value of a constant into a register? If the constant will fit into the 16-bit constant field of an instruction, we can use CMOVE to load the sign-extended constant into a register. This approach works for constants between -32768 and +32767. If the constant is too large, it's stored in a main memory location and we use a LD instruction to get the value into a register.

Loading the value of a variable is much the same as loading the value of a large constant: we use a LD instruction to access the memory location that holds the value of the variable.

Performing an array access is slightly more complicated: arrays are stored as consecutive locations in main memory, starting with index 0. Each array element occupies some fixed number of bytes. So we need code to convert the array index into the actual main memory address for the specified array element.

We first invoke compile_expr to generate code that evaluates the index expression and leaves the result in Rx. That will be a value between 0 and the size of the array minus 1. We'll use a LD instruction to access the appropriate array entry, but that means we need to convert the index into a byte offset, which we do by multiplying the index by bsize, the number of bytes in one element. If b was an array of integers, bsize would be 4. Now that we have the byte offset in a register, we can use LD to add the offset to the base address of the array computing the address of the desired array element, then load the memory value at that address into a register.

Assignment expressions are easy: invoke compile_expr to generate code that loads the value of the expression into a register, then generate a ST instruction to store the value into the specified variable.

Arithmetic operations are pretty easy too: use compile_expr to generate code for each of the operand expressions, leaving the results in registers. Then generate the appropriate ALU instruction to combine the operands and leave the answer in a register.

Compiling Expressions

C code:

```
int x, y;
y = (x-3)*(y+123456)
```

Beta assembly code:

```
x:  LONG(0)
y:  LONG(0)
c1: LONG(123456)
...

    LD(x, r1)
    CMOVE(3, r2)
    SUB(r1, r2, r1)      ⇒SUBC(r1,3,r1)
    LD(y, r2)
    LD(c1, r3)
    ADD(r2, r3, r2)
    MUL(r2, r1, r1)
    ST(r1, y)
```

```
compile_expr(y = (x-3)*(y+123456))
   compile_expr((x-3)*(y+123456))
      compile_expr(x-3)
         compile_expr(x)
            LD(x,r1)
         compile_expr(3)
            CMOVE(3,r2)
         SUB(r1,r2,r1)
      compile_expr(y+123456)
         compile_expr(y)
            LD(y,r2)
      compile_expr(123456)
            LD(c1,r3)
         ADD(r2,r3,r2)
   MUL(r1,r2,r1)
ST(r1,y)
```

Let's look at example to see how all this works. Here we have an assignment expression that requires a subtract, a multiply, and an addition to compute the required value.

Let's follow the compilation process from start to finish as we invoke compile_expr to generate the necessary code.

Following the template for assignment expressions from the previous page, we recursively call compile_expr to compute value of the right-hand-side of the assignment.

That's a multiply operation, so, following the Operations template, we need to compile the left-hand operand of the multiply.

That's a subtract operation, so, we call compile_expr again to compile the left-hand operand of the subtract.

Aha, we know how to get the value of a variable into a register. So we generate a LD instruction to load the value of x into r1.

The process we're following is called "recursive descent". We've used recursive calls to compile_expr to process each level of the expression tree. At each recursive call the expressions get simpler, until we reach a variable or constant, where we can generate the appropriate instruction without descending further. At this point we've reached a leaf of the expression tree and we're done with this branch of the recursion.

We now need to get the value of the right-hand operand of the subtract into a register. In case it's a small constant, so we generate a CMOVE instruction.

Now that both operand values are in registers, we return to the subtract template and generate a SUB instruction to do the subtraction. We now have the value for the left-hand operand of the multiply in r1.

We follow the same process for the right-hand operand of the multiply, recursively calling compile_expr to process each level of the expression until we reach a variable or constant. Then we return up the expression tree, generating the appropriate instructions as we go, following the dictates of the appropriate template from the previous slide.

The generated code is shown on the left of the slide. The recursive-descent technique makes short work of generating code for even the most complicated of expressions.

There's even opportunity to find some simple optimizations by looking at adjacent instructions. For example, a CMOVE followed by an arithmetic operation can often be shortened to a single arithmetic instruction with the constant as its second operand. These local transformations are called "peephole optimizations" since we're only considering just one or two instructions at a time.

compile_statement

- Unconditional: *expr;*

  Beta assembly:

  compile_expr(*expr*)

- Compound: { $statement_1$; $statement_2$; ... }

  Beta assembly:

  compile_statement($statement_1$)
  compile_statement($statement_2$)
  …

Now let's turn our attention to compile_statement.

The first two statement types are pretty easy to handle. Unconditional statements are usually assignment expressions or procedure calls. We'll simply ask compile_expr to generate the appropriate code.

Compound statements are equally easy. We'll recursively call compile_statement to generate code for each statement in turn. The code for statement_2 will immediately follow the code generated for statement_1. Execution will proceed sequentially through the code for each statement.

compile_statement: Conditional

C code:

```
if (expr)
  statement;
```

Beta assembly:

```
    compile_expr(expr)⇒Rx
    BF(rx, Lendif)
    compile_statement(statement)
Lendif:
```

C code:

```
if (expr)
  statement1;
else
  statement2;
```

Beta assembly:

```
    compile_expr(expr)⇒Rx
    BF(rx, Lelse)
    compile_statement(statement1)
    BR(Lendif)
Lelse:
    compile_statement(statement2)
Lendif:
```

Here we see the simplest form of the conditional statement, where we need to generate code to evaluate the test expression and then, if the value in the register is FALSE, skip over the code that executes the statement in the THEN clause. The simple assembly-language template uses recursive calls to compile_expr and compile_statement to generate code for the various parts of the IF statement.

The full-blown conditional statement includes an ELSE clause, which should be executed if the value of the test expression is FALSE. The template uses some branches and labels to ensure the course of execution is as intended.

You can see that the compilation process is really just the application of many small templates that break the code generation task down step-by-step into smaller and smaller tasks, generating the necessary code to glue all the pieces together in the appropriate fashion.

compile_statement: Iteration

C code:

```
while (expr)
   statement;
```

Beta assembly:

```
Lwhile:
   compile_expr(expr)⇒Rx
   BF(rx, Lendwhile)
   compile_statement(statement)
   BR(Lwhile)
Lendwhile:
```

Better Beta assembly:

```
   BR(Ltest)
Lwhile:
   compile_statement(statement)
Ltest:
   compile_expr(expr)⇒Rx
   BT(rx, Lwhile)
```

—Saves an instruction each iteration

C code:

```
for (init; test; increment)
  statement;
```

Example:

```
for (i=0; i < 10; i = i + 1)
  sum = sum + b[i];
```

is equivalent to:

```
init;
while (test) {
   statement;
   increment;
}
```

And here's the template for the WHILE statement, which looks a lot like the template for the IF

statement with a branch at the end that causes the generated code to be re-executed until the value of the test expression is FALSE.

With a bit of thought, we can improve on this template slightly. We've reorganized the code so that only a single branch instruction (BT) is executed each iteration, instead of the two branches (BF, BR) per iteration in the original template. Not a big deal, but little optimizations to code inside a loop can add up to big savings in a long-running program.

Just a quick comment about another common iteration statement, the FOR loop. The FOR loop is a shorthand way of expressing iterations where the loop index ("i" in the example shown) is run through a sequence of values and the body of the FOR loop is executed once for each value of the loop index.

The FOR loop can be transformed into the WHILE statement shown here, which can then be compiled using the templates shown above.

Putting It All Together: Factorial

```
int n = 20;          n: LONG(20)
int r = 0;           r: LONG(0)
                     start:
r = 1;                   CMOVE(1, r0)
                         ST(r0, r)
while (n > 0) {          BR(test)
                     loop:
    r = r*n;             LD(r, r3)
                         LD(n,r1)
                         MUL(r1, r3, r3)
                         ST(r3, r)
    n = n-1;             LD(n,r1)
                         SUBC(r1, 1, r1)
                         ST(r1, n)
}                    test:
                         LD(n, r1)
                         CMPLT(r31, r1, r2)
                         BT(r2, loop)
                     done:
```

Easy translation

Slow code
(10 instructions in the loop)

In this example, we've applied our templates to generate code for the iterative implementation of the factorial function that we've seen before. Look through the generated code and you'll be able to match code fragments with the templates from last couple of slides. It's not the most efficient code, but not bad given the simplicity of the recursive-descent approach for compiling high-level programs.

Optimization: keep values in regs

```
int n = 20,          n: LONG(20)
int r;               r: LONG(0)

r = 1;               start:
                        CMOVE(1, r0)
                        ST(r0, r)
                        LD(n,r1)    | keep n in r1
                        LD(r,r3)    | keep r in r3

                        BR(test)
while (n > 0)        loop:
 {                      MUL(r1, r3, r3)
    r = r*n;            SUBC(r1, 1, r1)
    n = n-1;         test:
```

Optimization:
Keep n, r in registers
⇒ move LDs/STs out of loop!

```
}                   CMPLT(r31, r1, r2)
                    BT(r2, loop)
```

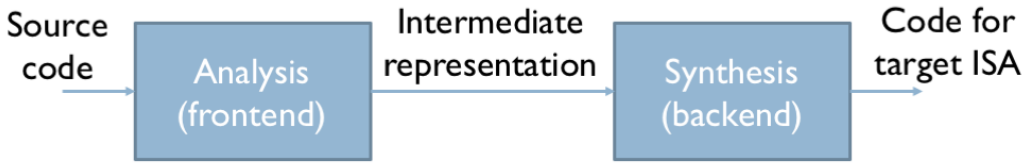
4 instructions in the loop

```
done:
    ST(r1,n)    | save final n
    ST(r3,r)    | save final r
```

It's a simple matter to modify the recursive-descent process to accommodate variable values that are stored in dedicated registers rather than in main memory. Optimizing compilers are quite good at identifying opportunities to keep values in registers and hence avoid the LD and ST operations needed to access values in main memory. Using this simple optimization, the number of instructions in the loop has gone from 10 down to 4. Now the generated code is looking pretty good!

But rather than keep tweaking the recursive-descent approach, let's stop here. In the next segment, we'll see how modern compilers take a more general approach to generating code. Still though, the first time I learned about recursive descent, I ran home to write a simple implementation and marveled at having authored my own compiler in an afternoon!

Anatomy of a Modern Compiler



- Read source program
- Break it up into basic elements
- Check correctness, report errors
- Translate to generic intermediate representation (IR)

- Optimize IR
- Translate IR to ASM
- Optimize ASM

A modern compiler starts by analyzing the source program text to produce an equivalent sequence of operations expressed in a language — and machine-independent intermediate representation (IR). The analysis, or frontend, phase checks that the program is well-formed, *i.e.*, that the syntax of each high-level language statement is correct. It understands the meaning (semantics) of each statement. Many high-level languages include declarations of the type — *e.g.*, integer, floating point, string, etc. — of each variable, and the frontend verifies that all operations are correctly applied, ensuring that numeric operations have numeric-type operands, string operations have string-type operands, and so on. Basically the analysis phase converts the text of the source program into an internal data structure that specifies the sequence and type of operations to be performed.

Often there are families of frontend programs that translate a variety of high-level languages (e.g, C, C++, Java) into a common IR.

The synthesis, or backend, phase then optimizes the IR to reduce the number of operations that will be executed when the final code is run. For example, it might find operations inside of a loop that are independent of the loop index and can moved outside the loop, where they are performed once instead of repeatedly inside the loop. Once the IR is in its final optimized form, the backend generates code sequences for the target ISA and looks for further optimizations that take advantage of particular features of the ISA. For example, for the Beta ISA we saw how a CMOVE followed by an arithmetic operation can be shorted to a single operation with a constant operand.

Frontend Stages

- Lexical analysis (scanning): Source → List of tokens

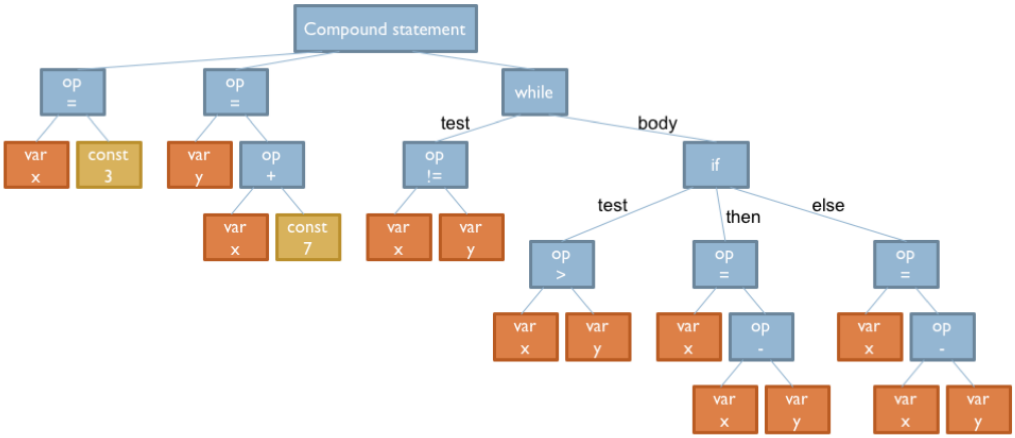
```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
    y = y - x;
  }
}
```

```
("int", KEYWORD)
("x", IDENTIFIER)
("=", OPERATOR)
("3", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("int", KEYWORD)
("y", IDENTIFIER)
("=", OPERATOR)
("x", IDENTIFIER)
("+", OPERATOR)
("7", INT_CONSTANT)
(";", SPECIAL_SYMBOL)
("while", KEYWORD)
("(", SPECIAL_SYMBOL)
...
```

The analysis phase starts by scanning the source text and generating a sequence of token objects that identify the type of each piece of the source text. While spaces, tabs, newlines, and so on were needed to separate tokens in the source text, they've all been removed during the scanning process. To enable useful error reporting, token objects also include information about where in the source text each token was found, *e.g.*, the file name, line number, and column number. The scanning phase reports illegal tokens, *e.g.*, the token "3x" would cause an error since in C it would not be a legal number or a legal variable name.

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree



The parsing phase processes the sequence of tokens to build the syntax tree, which captures the structure of the original program in a convenient data structure. The operands have been organized for each unary and binary operation. The components of each statement have been found and labeled. The role of each source token has been determined and the information captured in the

syntax tree.

Compare the labels of the nodes in the tree to the templates we discussed in the previous segment. We can see that it would be easy to write a program that did a depth-first tree walk, using the label of each tree node to select the appropriate code generation template. We won't do that quite yet since there's still some work to be done analyzing and transforming the tree.

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree
- Semantic analysis (mainly, type checking)

Consider:

```
int x = "bananas";
```

Syntax OK
Semantically (meaning) WRONG

Var	Type
x	int

```
Line 1: error, invalid conversion from string constant to int
```

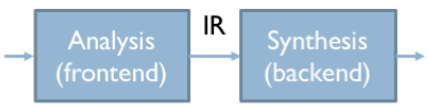
The syntax tree makes it easy to verify that the program is semantically correct, *e.g.*, to check that the types of the operands are compatible with the requested operation.

For example, consider the statement x = "bananas". The syntax of the assignment operation is correct: there's a variable on the left-hand side and an expression on the right-hand side. But the semantics is not correct, at least in the C language! By looking in its symbol table to check the declared type for the variable x (int) and comparing it to the type of the expression (string), the semantic checker for the "op =" tree node will detect that the types are not compatible, *i.e.*, that we can't store a string value into an integer variable.

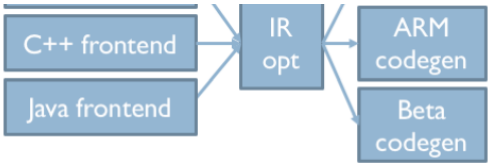
When the semantic analysis is complete, we know that the syntax tree represents a syntactically correct program with valid semantics, and we've finished converting the source program into an equivalent, language-independent sequence of operations.

Intermediate Representation (IR)

- Internal compiler language that is:
  - Language-independent
  - Machine-independent
  - Easy to optimize



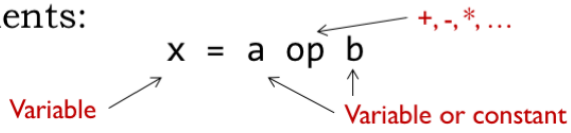
- Why yet another language?
  - Assembly does not have enough info to optimize it well
  - Enables modularity and reuse



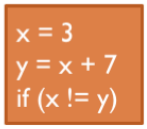
The syntax tree is a useful intermediate representation (IR) that is independent of both the source language and the target ISA. It contains information about the sequencing and grouping of operations that isn't apparent in individual machine language instructions. And it allows frontends for different source languages to share a common backend targeting a specific ISA. As we'll see, the backend processing can be split into two sub-phases. The first performs machine-independent optimizations on the IR. The optimized IR is then translated by the code generation phase into sequences of instructions for the target ISA.

Common IR: Control Flow Graph

- Assignments:



- Basic block: Sequence of assignments with an optional branch at the end



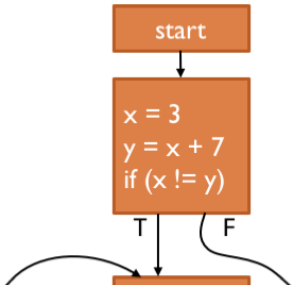
- Control flow graph:
  - Nodes: Basic blocks
  - Edges: branches between basic blocks

A common IR is to reorganize the syntax tree into what's called a control flow graph (CFG). Each node in the graph is a sequence of assignment and expression evaluations that ends with a branch. The nodes are called "basic blocks" and represent sequences of operations that are executed as a unit: once the first operation in a basic block is performed, the remaining operations will also be performed without any other intervening operations. This knowledge lets us consider many optimizations, *e.g.*, temporarily storing variable values in registers, that would be complicated if there was the possibility that other operations outside the block might also need to access the variable values while we were in the middle of this block.

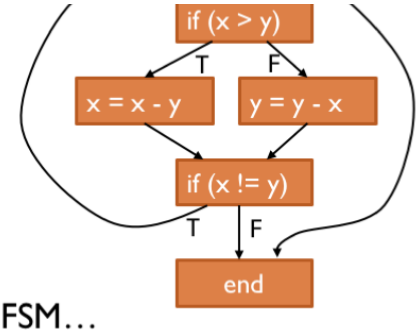
The edges of the graph indicate the branches that take us to another basic block.

Control Flow Graph for GCD

```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
```



```
  } else {
    y = y - x;
  }
}
```



Looks like a high-level FSM…

For example, here's the CFG for GCD.

If a basic block ends with a conditional branch, there are two edges, labeled "T" and "F" leaving the block that indicate the next block to execute depending on the outcome of the test. Other blocks have only a single departing arrow, indicating that the block always transfers control to the block indicated by the arrow.

Note that if we can arrive at a block from only a single predecessor block, then any knowledge we have about operations and variables from the predecessor block can be carried over to the destination block. For example, if the "if (x > y)" block has generated code to load the values of x and y into registers, both destination blocks can use that information and use the appropriate registers without having to generate their own LDs.

But if a block has multiple predecessors, such optimizations are more constrained: we can only use knowledge that is common to *all* the predecessor blocks.

The CFG looks a lot like the state transition diagram for a high-level FSM!

IR Optimization

- Perform a set of passes over the CFG
  - Each pass does a specific, simple task over the CFG
  - By repeating multiple simple passes on the CFG over and over, compilers achieve very complex optimizations
- Example optimizations:
  - Dead code elimination: Eliminate assignments to variables that are never used, or basic blocks that are never reached
  - Constant propagation: Identify variables that are constant, substitute the constant elsewhere
  - Constant folding: Compute and substitute constant expressions

We'll optimize the IR by performing multiple passes over the CFG. Each pass performs a specific, simple optimization. We'll repeatedly apply the simple optimizations in multiple passes, until the we can't find any further optimizations to perform. Collectively, the simple optimizations can combine to achieve very complex optimizations.

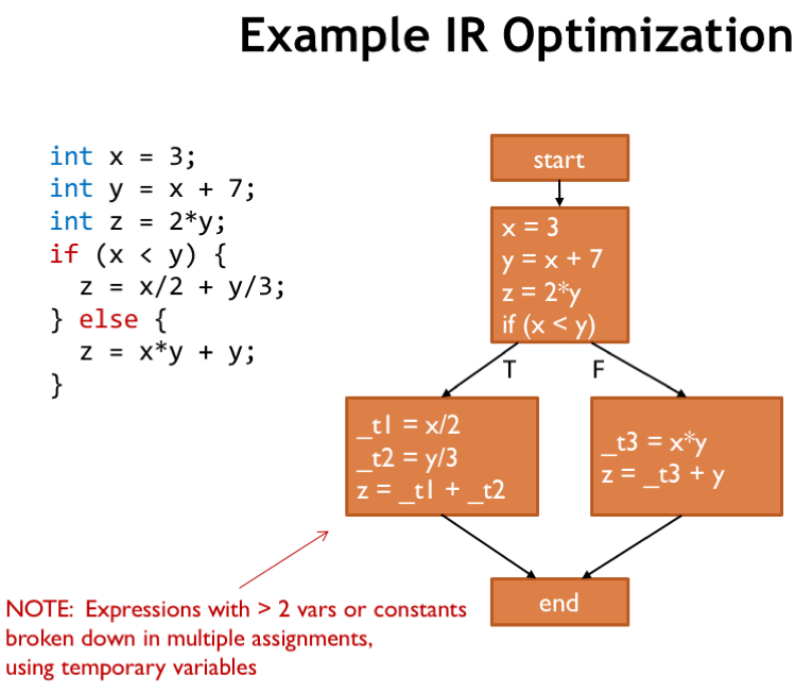
Here are some example optimizations:

We can eliminate assignments to variables that are never used and basic blocks that are never

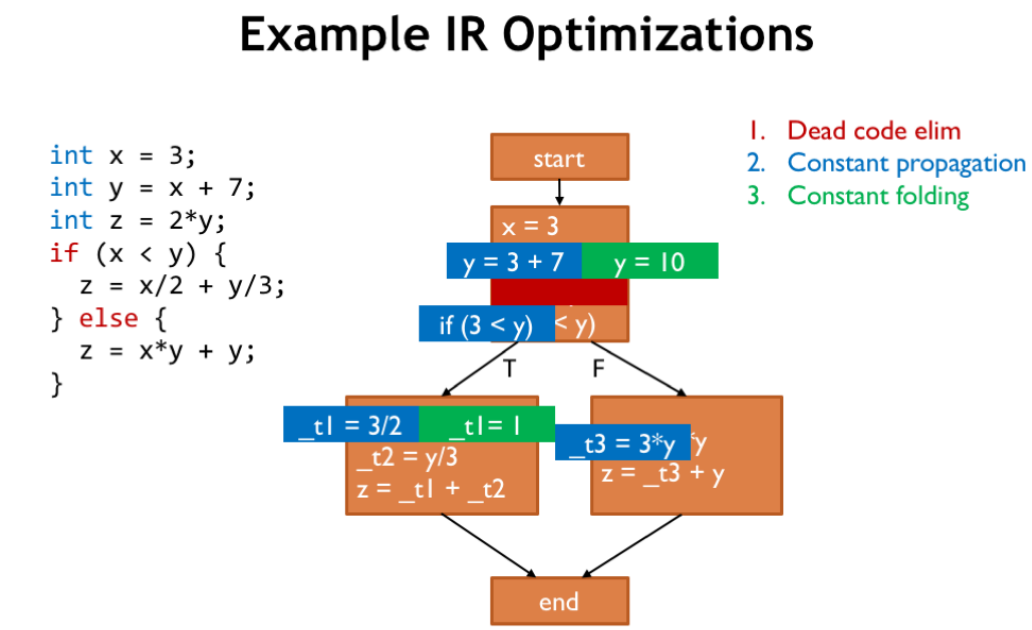
reached. This is called “dead code elimination”.

In constant propagation, we identify variables that have a constant value and substitute that constant in place of references to the variable.

We can compute the value of expressions that have constant operands. This is called “constant folding”.



To illustrate how these optimizations work, consider this slightly silly source program and its CFG. Note that we’ve broken down complicated expressions into simple binary operations, using temporary variable names (e.g, “_t1”) to name the intermediate results.



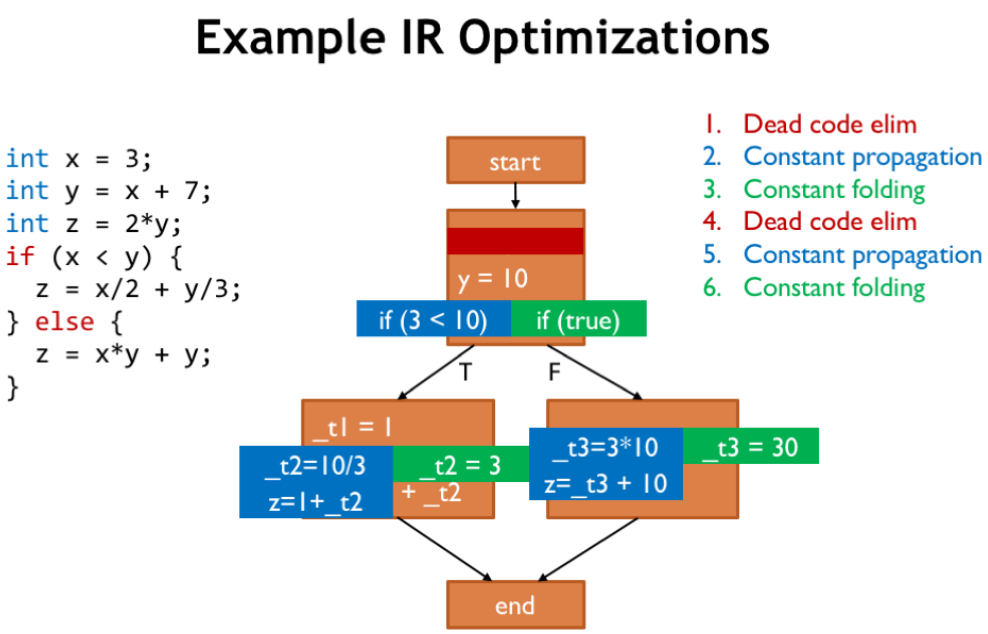
Let’s get started!

The dead code elimination pass can remove the assignment to Z in the first basic block since Z is

The dead code elimination pass can remove the assignment to Z in the first basic block since Z is reassigned in subsequent blocks and the intervening code makes no reference to Z.

Next we look for variables with constant values. Here we find that X is assigned the value of 3 and is never re-assigned, so we can replace all references to X with the constant 3.

Now perform constant folding, evaluating any constant expressions.

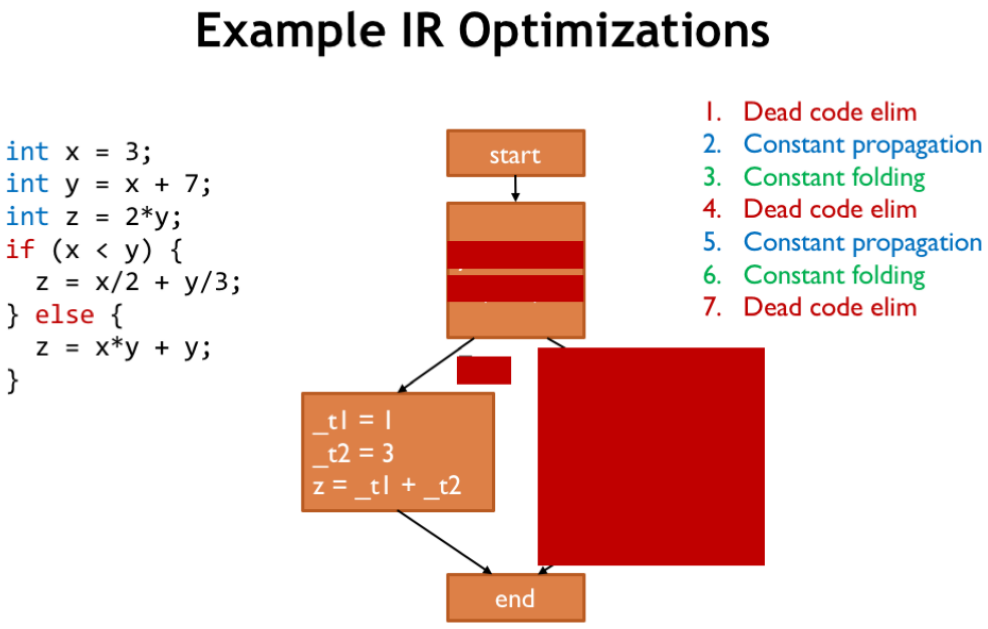


Here's the updated CFG, ready for another round of optimizations.

First dead code elimination.

Then constant propagation.

And, finally, constant folding.

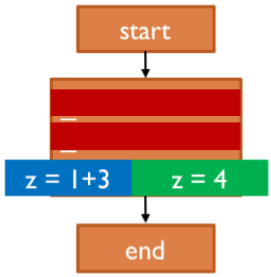


So after two rounds of these simple operations, we've thinned out a number of assignments. On to round three!

Dead code elimination. And here we can determine the outcome of a conditional branch, eliminating entire basic blocks from the IR, either because they're now empty or because they can no longer be reached.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim

Wow, the IR is now considerably smaller.

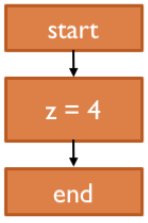
Next is another application of constant propagation.

And then constant folding.

Followed by more dead code elimination.

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
  z = x/2 + y/3;
} else {
  z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
11. Constant propagation
12. Constant folding
13. Dead code elim
14. Constant propagation
15. Constant folding

No changes in 13,14, 15 → DONE

Dumb repetition of simple transformations on CFGs

Extremely powerful optimizations

More optimizations by adding passes: Common subexpression elimination, loop-invariant code motion, loop unrolling…

The passes continue until we discover there are no further optimizations to perform, so we're done!

Repeated applications of these simple transformations have transformed the original program into an equivalent program that computes the same final value for Z.

We can do more optimizations by adding passes: eliminating redundant computation of common subexpressions, moving loop-independent calculations out of loops, unrolling short loops to perform the effect of, say, two iterations in a single loop execution, saving some of the cost of increment and test instructions. Optimizing compilers have a sophisticated set of optimizations they employ to make smaller and more efficient code.

Code Generation

- Translate generated IR to assembly
- Register allocation: Map variables to registers
  - If variables > registers, map some to memory, and load/store them when needed
- Translate each assignment to instructions
  - Some assignments may require > 1 instr if our ISA doesn't have op
- Emit each basic block: label, assignments, and branches
- Lay out basic blocks, removing superfluous jumps
- ISA and CPU-specific optimizations
  - e.g., if possible, reorder instructions to improve performance

Okay, we're done with optimizations. Now it's time to generate instructions for the target ISA.

First the code generator assigns each variable a dedicated register. If we have more variables than registers, some variables are stored in memory and we'll use LD and ST to access them as needed. But frequently-used variables will almost certainly live as much as possible in registers.

Use our templates from before to translate each assignment and operation into one or more instructions.

Then emit the code for each block, adding the appropriate labels and branches.

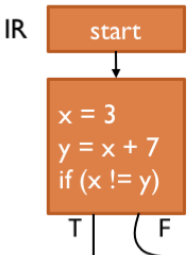
Reorder the basic block code to eliminate unconditional branches wherever possible.

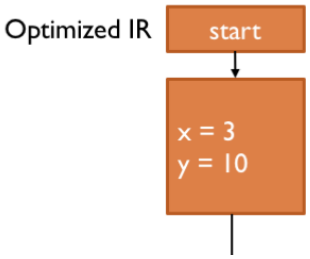
And finally perform any target-specific peephole optimizations.

Putting It All Together: GCD

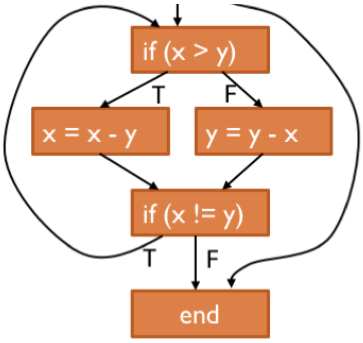
Source code

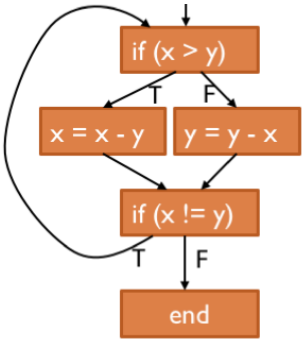
```
int x = 3;
int y = x + 7;
while (x != y) {
  if (x > y) {
    x = x - y;
  } else {
```





```
    y = y - x;
  }
}
```

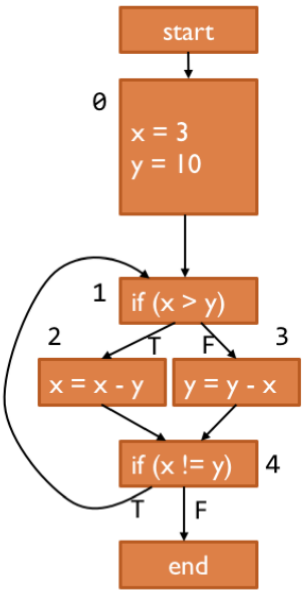




Here's the original CFG for the GCD code, along with the slightly optimized CFG. GCD isn't as trivial as the previous example, so we've only been able to do a bit of constant propagation and constant folding.

Note that we can't propagate knowledge about variable values from the top basic block to the following "if" block since the "if" block has multiple predecessors.

Putting It All Together: GCD



1. Allocate registers:
x: R0, y: R1

2. Produce each basic block:

```
BBL0:   CMOVE(3, R0)
        CMOVE(10, R1)
        BR(BBL1)

BBL1:   CMPLT(R1, R0, R2)
        BT(R2, BBL2)
        BR(BBL3)

BBL2:   SUB(R0, R1, R0)
        BR(BBL4)

BBL3:   SUB(R1, R0, R1)
        BR(BBL4)

BBL4:   CMPEQ(R1, R0, R2)
        BT(R2, end)
        BR(BBL1)
end:
```

3. Lay out BBs, removing superfluous branches:

```
BBL0:   CMOVE(3, R0)
        CMOVE(10, R1)
BBL1:   CMPLT(R1, R0, R2)
        BT(R2, BBL2)
BBL3:   SUB(R1, R0, R1)
        BR(BBL4)
BBL2:   SUB(R0, R1, R0)
BBL4:   CMPEQ(R1, R0, R2)
        BF(R2, BBL1)
end:
```

Here's how the code generator will process the optimized CFG.

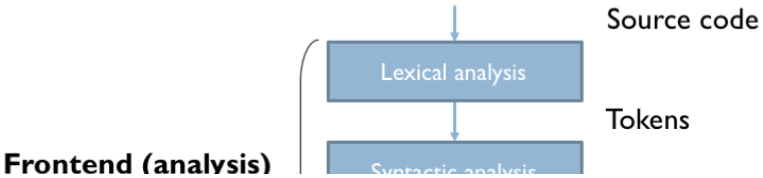
First, it dedicates registers to hold the values for x and y.

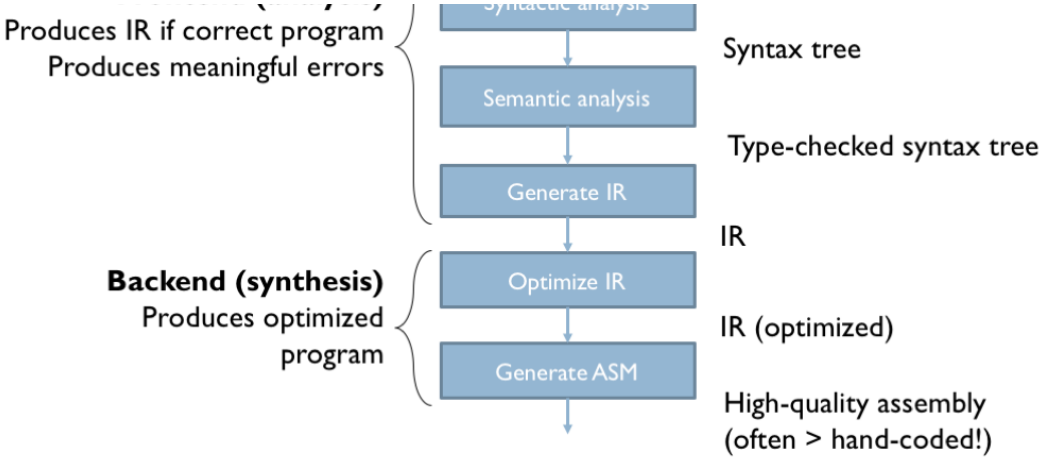
Then, it emits the code for each of the basic blocks.

Next, reorganize the order of the basic blocks to eliminate unconditional branches wherever possible.

The resulting code is pretty good. There are no obvious changes that a human programmer might make to make the code faster or smaller. Good job, compiler!

Summary: Modern Compilers





Here are all the compilation steps shown in order, along with their input and output data structures. Collectively they transform the original source code into high-quality assembly code. The patient application of optimization passes often produces code that's more efficient than writing assembly language by hand.

Nowadays, programmers are able to focus on getting the source code to achieve the desired functionality and leave the details of translation to instructions in the hands of the compiler.

- BackCompilers
- ContinueTopic Videos

Computation Structures

Compilation Worksheet

compile_expr(expr) ⇒ Rx

- Constants: `1234` ⇒ Rx
  - `CMOVE(1234,Rx)`
  - `LD(c1,Rx)`
    …
    `c1: LONG(123456)`
- Variables: `a` ⇒ Rx
  - `LD(a,Rx)`
    `...`
    `a: LONG(0)`
- Variables: `b[expr]` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `MULC(Rx,bsize,Rx)`
    `LD(Rx,b,Rx)`
    …
    `// reserve array space`
    `b: . = . + bsize*blen`
- Operations: $expr_1$ + $expr_2$ ⇒ Rx
  - `compile_expr(`$expr_1$`)⇒Rx`
    `compile_expr(`$expr_2$`)⇒Ry`
    `ADD(Rx,Ry,Rx)`
- Procedure call: `f(`e_1, e_2, …`)` ⇒ Rx
  next lecture!
- Assignment: `a=expr` ⇒ Rx
  - `compile_expr(expr)⇒Rx`
    `ST(Rx,a)`

compile_statement(…)

- Unconditional: *expr*;
  - `compile_expr(expr)`
- Compound: { *s1; s2;* … }
  - `compile_statement(s1)`
    `compile_statement(s2)`
    …
- Conditional: `if (expr) s1;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lendif)`
    `compile_statement(s1)`
    `Lendif:`
- Conditional: `if (expr) s1; else s2;`
  - `compile_expr(expr)⇒Rx`
    `BF(Rx,Lelse)`
    `compile_statement(s1)`
    `BR(Lendif)`
    `Lelse:`
    `compile_statement(s1)`
    `Lendif:`
- Iteration: while (*expr*) *s1*;
  - `BR(Ltest)`
    `Lwhile:`
    `compile_statement(s1)`
    `Ltest:`
    `compile_expr(expr)⇒Rx`
    `BT(Rx, Lwhile)`
- Iteration: `for (init; test; incr) s1;`
  init;
  while (*test*) { *s1; incr;* }

Problem 1.

Please hand-compile the following snippets of C code into equivalent Beta assembly language statements. Assume that memory locations have been allocated for the all C variables with labels that corresponds to the variable names. So to load the value of the C variable `a` into register `R3`, the appropriate assembly language statement would be `LD(R31,a,R3)`. And to store the value in `R17` to the C variable `b`, the appropriate assembly language statement would be `ST(R17,b,R31)`. Similarly, assume that memory locations have been allocated for each C array, with a label defined whose value is the address of the 0th element of the array.

(A)
```
a = 42;
```
CMOVE(42,R0)
ST(R0,a)

(B)
```
c = 5*x - 13;
```
LD(x,R0) ST(R0,c)
MUL(R0,5,R0)
SUB(R0,13,R0)

(C)
```
y = (x - 3)*(y + 123456);
```
LD(x,R0) MUL(R0,R1,R0)
LD(y,R1) ST(R0,y)
SUBC(R0,3,R0) C: 123456
ADDC(R1,C1,R1)

(D)
```
if (a == 3) b = b + 1;
```
LD(a,R0)
SUBC(R0,3,R0)
BF(R0,Lendif)
LD(b,R0)
ADDC(R0,1,R0)
ST(R0,b)
Lendif:

(E)
```
a[i] = a[i-1];
```
LD(a,R0) ADDC(R0,4,R0)
LD(i,R1) ST(R1,R0)
MULC(R1,4,R1)
ADDC(R0,R1,R0)
SUBC(R0,4,R0)
LD(R0,R1)

(F)
```
x = y[3] + y[12];
```
LD(y,12,R0) ST(R0,x)
LDC(y,48,R1)
ADD(R0,R1,R0)

(G)
```
if (b == 0 || b < min) {
    min = b;
} else {
    too_big += 1;
}
```
LD(b,R0) ST(R0,min)
CMPEQC(R0,0,R1) BR(L2)
LD(min,R2) L1:
CMPLT(R0,R2,R1) LDC(too_big,R0)
BF(R1,L1) ADDC(R0,1,R0)
ST(R0,too_big)
L2:

(H)
```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i
    i = i + 1;
}
```
CMOVEC(0,R0)
CMOVE(0,R1)
BR(L2)
L1:
ADD(R0,R1,R0)
ADDC(R1,1,R1)
L2:
CMPLTC(R1,10,R2)
BT(R2,L1)
ST(R0,sum) ST(R1,i)

Problem 2.

In block-structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```

A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

```
      CMOVE(0,R0)    ADDC(R1,1,R1)    ST(R2,SUM)
L:>   CMOVE(0,R1)    CMPLTC(R1,10,R2)
      ADD(R0,R1,R0)  BT(R2,L)
```

B. Define a *memory access* as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part G of the preceding problem).

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) {
      sum += i; sum += i+1;
  }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

Problem 3.

Which of the following Beta instruction sequences might have resulted from compiling the following C statement? For each sequence describe the value that does end up as the value of y.

```
int x[20], y;
y = x[1] + 4;
```

A.
```
LD (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```
R0 = x+5

Program exception

B.
```
CMOVE (4, R0)
ADDC (R0, x + 4, R0)
ST (R0, y, R31)
```
R0 = 4

R0 = x+8

Y = x[2]

C.
```
LD (R31, x + 4, R0)
ST (R0, y + 4, R31)
```
R0 = x[1]

D.
```
CMOVE (4, R0)
LD (R0, x, R1)
ST (R1, y, R0)
```
R0 = 4

R1 = x[1]

E.
```
LD (R31, x + 4, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```
R0 = x[1]

F.
```
ADDC (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```
R0 = x+1

Problem 1.

Please hand-compile the following snippets of C code into equivalent Beta assembly language statements. Assume that memory locations have been allocated for the all C variables with labels that corresponds to the variable names. So to load the value of the C variable `a` into register `R3`, the appropriate assembly language statement would be `LD(R31,a,R3)`. And to store the value in `R17` to the C variable `b`, the appropriate assembly language statement would be `ST(R17,b,R31)`. Similarly, assume that memory locations have been allocated for each C array, with a label defined whose value is the address of the 0th element of the array.

(A)
```
a = 42;
```

```
CMOVE(42,R0)
ST(R0,a,R31)
```

(B)
```
c = 5*x - 13;
```

```
CMOVE(5,R0)
LD(x,R1)
MUL(R0,R1,R0)
CMOVE(13,R1)
SUB(R0,R1,R0)
ST(R0,c)
```

Optimized:

```
LD(x,R1)
MULC(R0,5,R0)
SUBC(R0,13,R0)
ST(R0,c)
```

(C)
```
y = (x - 3)*(y + 123456);
```

```
LD(x,R0)
SUBC(R0,3,R0)
LD(y,R1)
LD(const,R2)
ADD(R1,R2,R1)
MUL(R0,R1,R0)
ST(R0,y)
```

```
// mem locn to hold
// large constant
const: LONG(123456)
```

(D)
```
if (a == 3) b = b + 1;
```

```
LD(R31,a,R0)
CMPEQC(R0,3,R0S)
BF(R0,L1)
LD(R31,b,R0)
ADDC(R0,1,R0)
ST(R0,b,R31)
L1:
```

(E)
```
a[i] = a[i-1];
```

```
LD(i,R0)
MULC(R0,4,R0) // 4 bytes/element
LD(R0,a-4,R1) // sub 4 at assy time!
ST(R1,a,R0)
```

(F)
```
x = y[3] + y[12];
```

```
LD(y+4*3,R0)
LD(y+4*12,R1)
ADD(R0,R1,R0)
ST(R0,x)
```

(G)
```
if (b == 0 || b < min) {
    min = b;
} else {
    too_big += 1;
}
```

```
LD(B,R0)
BEQ(R0,L2)
LD(min,R1)
CMPLT(R0,R1,R1)
BF(R1,L3)
L2:
ST(R0,min)
BR(L4)
L3:
LD(too_big,R0)
ADDC(R0,1,R0)
ST(R0,too_big)
L4:
```

(H)
```
sum = 0;
i = 0;
while (i < 10) {
    sum = sum + i
    i = i + 1;
}
```

Unoptimized:

```
ST(R31,sum)
ST(R31,i)
L5:
LD(sum,R0)
LD(i,R1)
ADD(R0,R1,R0)
ST(R0,sum)
LD(i,R0)
ADDC(R0,1,R0)
ST(R0,i)
L6:
LD(i,R0)
CMPLTC(R0,10,R1)
BT(R2,L5)
```

Problem 2.

In block-structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```

A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

B. Define a *memory access* as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part H of the preceding problem).

```
Part (H): each iteration
  10 instruction fetches
  6 data memory accesses
10 iterations => 160 accesses
+ 4 from first two insts => 164
```

```
Part (A): each iteration
  4 instruction fetches
  0 data memory accesses
10 iterations => 40 accesses
+ 6 from before/after loop => 46
```

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) {
      sum += i; sum += i+1;
  }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

```
  MOVE(R31,R2)  // sum
  ST(R2,sum)
  MOVE(R31,R1)  // i
L7:
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  ADD(R2,R1,R2)
  ADDC(R1,1,R1)
  CMPLTC(R1,10,R0)
  BT(R0,L7)
  ST(R2,sum)
```

```
Part (C): each iteration
  6 instruction fetches
  0 data memory accesses
5 iterations => 30 accesses
+ 6 from before/after loop => 36
```

Problem 3.

Which of the following Beta instruction sequences might have resulted from compiling the following C statement? For each sequence describe the value that does end up as the value of y.

```
int x[20], y;
y = x[1] + 4;
```

A.
```
LD (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. If x[0] is stored a location x, x[1] is stored at location x+4 since array element takes one word (4 bytes). Program exception trying to read from a location whose address is not a multiple of 4.

B.
```
CMOVE (4, R0)
ADDC (R0, x + 4, R0)
ST (R0, y, R31)
```

Not this one. The second instruction adds the *address* of x[1] to R0, not the contents of x[1].

C.
```
LD (R31, x + 4, R0)
ST (R0, y + 4, R31)
```

Not this one. This stores x[1] in the location *following* the one word of storage allocated for y.

D.
```
CMOVE (4, R0)
LD (R0, x, R1)
ST (R1, y, R0)
```

Not this one. It's equivalent to (C).

E.
```
LD (R31, x + 4, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Yes!!! This one…

F.
```
ADDC (R31, x + 1, R0)
ADDC (R0, 4, R0)
ST (R0, y, R31)
```

Not this one. The ADDC instruction loads the address of x plus 1 into R0, then increments it by 4. So y gets the value "x+5" where x is address of the 0^{th} element of the array.

12 Procedures and Stacks

12.1 ANNOTATED SLIDES

← BROWSE COURSE MATERIAL

- « Procedures and Stacks
- 12.1.1Annotated slides
- » Topic Videos

.toc { margin-left: 2em; } .lecslide { margin-top: 1em; margin-bottom: 1em; border-top: 0.5px solid #808080; padding-top: 1em; text-align: center; } .lecslideimg { width: 5in; border: 1px solid black; }

L12: Procedures and Stacks

Content of the following slides is described in the surrounding text.

Procedures: A Software Abstraction

- Procedure: Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal parameters
  - Local storage
  - Returns control to the caller

```
int gcd(int a, int b) {
  int x = a;
  int y = b;
  while (x != y) {
    if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
```

when finished

- Using multiple procedures enables abstraction and reuse
  - Compose large programs from collections of simple procedures

```
  return x;
}

bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}

coprimes(5, 10); // false
coprimes(9, 10); // true
```

One of the most useful abstractions provided by high-level languages is the notion of a procedure or subroutine, which is a sequence of instructions that perform a specific task.

A procedure has a single named entry point, which can be used to refer to the procedure in other parts of the program. In the example here, this code is defining the GCD procedure, which is declared to return an integer value.

Procedures have zero or more formal parameters, which are the names the code inside the procedure will use to refer the values supplied when the procedure is invoked by a "procedure call". A procedure call is an expression that has the name of the procedure followed by parenthesized list of values called "arguments" that will be matched up with the formal parameters. For example, the value of the first argument will become the value of the first formal parameter while the procedure is executing.

The body of the procedure may define additional variables, called "local variables", since they can only be accessed by statements in the procedure body. Conceptually, the storage for local variables only exists while the procedure is executing. They are allocated when the procedure is invoked and deallocated when the procedure returns.

The procedure may return a value that's the result of the procedure's computation. It's legal to have procedures that do not return a value, in which case the procedures would only be executed for their "side effects", *e.g.*, changes they make to shared data.

Here we see another procedure, COPRIMES, that invokes the GCD procedure to compute the greatest common divisor of two numbers. To use GCD, the programmer of COPRIMES only needed to know the input/output behavior of GCD, *i.e.*, the number and types of the arguments and what type of value is returned as a result. The procedural abstraction has hidden the implementation of GCD, while still making its functionality available as a "black box".

This is a very powerful idea: encapsulating a complex computation so that it can be used by others. Every high-level language comes with a collection of pre-built procedures, called "libraries", which can be used to perform arithmetic functions (*e.g.*, square root or cosine), manipulate collections of data (*e.g.*, lists or dictionaries), read data from files, and so on — the list is nearly endless! Much of the expressive power and ease-of-use provided by high-level languages comes from their libraries of "black boxes".

The procedural abstraction is at the heart of object-oriented languages, which encapsulate data and procedures as black boxes called objects that support specific operations on their internal data. For example, a LIST object has procedures (called "methods" in this context) for indexing into the list to read or change a value, adding new elements to the list, inquiring about the length of the list, and so on. The internal representation of the data and the algorithms used to implement the methods are hidden by the object abstraction. Indeed, there may be several different LIST implementations to choose from depending on which operations you need to be particularly efficient.

Okay, enough about the virtues of the procedural abstraction! Let's turn our attention to how to implement procedures using the Beta ISA.

Implementing Procedures

- Option 1: Inlining

- Option 1: Inlining
  - Compiler substitutes procedure call with body
  - Problems?
    - Code size
    - Recursion

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}
```

- Option 2: Linking
  - Produce separate code for each procedure
  - Caller evaluates input arguments, stores them and transfers control to the callee's entry point
  - Callee runs, stores result, transfers control to caller

A possible implementation is to "inline" the procedure, where we replace the procedure call with a copy of the statements in the procedure's body, substituting argument values for references to the formal parameters. In this approach we're treating procedures very much like UASM macros, *i.e.*, a simple notational shorthand for making a copy of the procedure's body.

Are there any problems with this approach? One obvious issue is the potential increase in the code size. For example, if we had a lengthy procedure that was called many times, the final expanded code would be huge! Enough so that inlining isn't a practical solution except in the case of short procedures where optimizing compilers do sometimes decide to inline the code.

A bigger difficulty is apparent when we consider a recursive procedure where there's a nested call to the procedure itself. During execution the recursion will terminate for some values of the arguments and the recursive procedure will eventually return an answer. But at compile time, the inlining process would not terminate and so the inlining scheme fails if the language allows recursion.

The second option is to "link" to the procedure. In this approach there is a single copy of the procedure code which we arrange to be run for each procedure call — all the procedure calls are said to link to the procedure code.

Here the body of the procedure is translated once into Beta instructions and the first instruction is identified as the procedure's entry point. The procedure call is compiled into a set of instructions that evaluate the argument expressions and save the values in an agreed-upon location. Then we'll use a BR instruction to transfer control to the entry point of the procedure. Recall that the BR instruction not only changes the PC but saves the address of the instruction following the branch in a specified register. This saved address is the "return address" where we want execution to resume when procedure execution is complete.

After branching to the entry point, the procedure code runs, stores the result in an agreed-upon location and then resumes execution of the calling program by jumping to the supplied return address.

Procedure Calling Convention

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}

fact(3);
```

```
fact(3) = 3*fact(2)
fact(2) = 2*fact(1)
fact(1) = 1*fact(0)
fact(0) = 1
```

fact(3);

- Need calling convention: Uniform way to transfer data and control between procedures
- Proposed convention:
  - Pass argument (value of n) in R1
  - Pass return address in R28
    - use `BR(fact,r28)` to call and `JMP(r28)` to return
  - Return result in R0

To complete this implementation plan we need a "calling convention" that specifies where to store the argument values during procedure calls and where the procedure should store the return value. It's tempting to simply allocate specific memory locations for the job: how about using registers? We could pass the argument value in registers starting, say, with R1. The return address could be stored in another register, say R28. As we can see, with this convention the BR and JMP instructions are just what we need to implement procedure call and return. It's usual to call the register holding the return address the "linkage pointer". And finally the procedure can use, say, R0 to hold the return value.

Let's see how this would work when executing the procedure call fact(3). As shown on the right, fact(3) requires a recursive call to compute fact(2), and so on. Our goal is to have a uniform calling convention where all procedure calls and procedure bodies use the same convention for storing arguments, return addresses and return values. In particular, we'll use the same convention when compiling the recursive call fact(n-1) as we did for the initial call to fact(3).

Procedure Linkage: First Try

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n - 1);
  } else {
    return 1;
  }
}

fact(3);
```

```
fact:
        CMPLEC(r1,0,r0)
        BT(r0,else)
        MOVE(r1,r2) // save n
        SUBC(r2,1,r1)
        BR(fact,r28)
        MUL(r0,r2,r0)
        BR(rtn)
else:   CMOVE(1,r0)
rtn:    JMP(r28)

main:   CMOVE(3,r1)
        BR(fact,r28)
        HALT()
```

OOPS!

- Proposed convention:
  - Pass argument (value of n) in R1
  - Pass return address in R28
  - Return result in R0

Okay. In the code shown on the right we've used our proposed convention when compiling the Beta code for fact(). Let's take a quick tour.

To compile the initial call fact(3) the compiler generated a CMOVE instruction to put the argument value in R1 and then a BR instruction to transfer control to fact's entry point while remembering the return address in R28.

The first statement in the body of fact tests the value of the argument using CMPLEC and BT instructions.

When n is greater than 0, the code performs a recursive call to fact, saving the value of the recursive argument n-1 in R1 as our convention requires. Note that we had to first save the value of the original

argument n because we'll need it for the multiplication after the recursive call returns its value in R0.

If n is not greater than 0, the value 1 is placed in R0. Then the two possible execution paths merge, each having generated the appropriate return value in R0, and finally there's a JMP to return control to the caller. The JMP instruction knows to find the return address in R28, just where the BR put it as part of the original procedure call.

Some of you may have noticed that there are some difficulties with this particular implementation. The code is correct in the sense that it faithfully implements procedure call and return using our proposed convention. The problem is that during recursive calls we'll be overwriting register values we need later.

For example, note that following our calling convention, the recursive call also uses R28 to store the return address. When executed, the code for the original call stored the address of the HALT instruction in R28. Inside the procedure, the recursive call will store the address of the MUL instruction in R28. Unfortunately that overwrites the original return address.

Even the attempt to save the value of the argument N in R2 is doomed to fail since during the execution of the recursive call R2 will be overwritten.

The crux of the problem is that each recursive call needs to remember the value of its argument and return address, *i.e.*, we need two storage locations for each active call to fact(). And while executing fact(3), when we finally get to calling fact(0) there are four nested active calls, so we'll need 4*2 = 8 storage locations. In fact, the amount of storage needed varies with the depth of the recursion. Obviously we can't use just two registers (R2 and R28) to hold all the values we need to save.

One fix is to disallow recursion! And, in fact, some of the early programming languages such as FORTRAN did just that. But let's see if we can solve the problem another way.

Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results
- Local storage:
  - Variables that compiler can't fit in registers
  - Space to save caller's register values for registers that we overwrite

Each of these is specific to a particular *activation* of a procedure. We call them the procedure's *activation record*

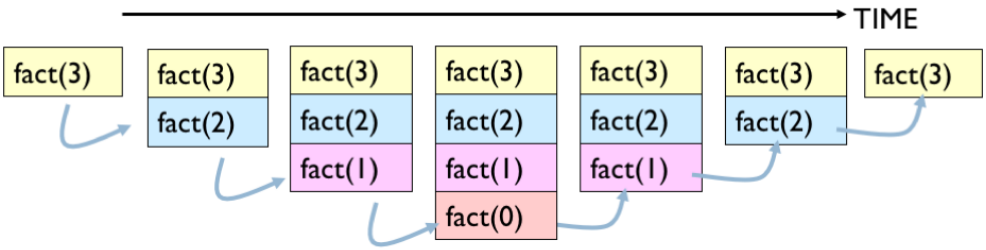
The problem we need to solve is where to store the values needed by procedure: its arguments, its return address, its return value. The procedure may also need storage for its local variables and space to save the values of the caller's registers before they get overwritten by the procedure. We'd like to avoid any limitations on the number of arguments, the number of local variables, etc.

So we'll need a block of storage for each active procedure call, what we'll call the "activation record". As we saw in the factorial example, we can't statically allocate a single block of storage for a particular procedure since recursive calls mean we'll have many active calls to that procedure at points during the execution.

What we need is a way to dynamically allocate storage for an activation record when the procedure is called, which can then be reclaimed when the procedure returns.

Activation Records

```
int fact(int n) {
  if (n > 0) return n*fact(n - 1);
  else return 1;
}
```



A procedure call creates a new activation record. Caller's record is preserved because we'll need it when callee finally returns.

Return to previous activation record when procedure finishes, permanently discarding activation record created by call we are returning from.

Let's see how activation records come and go as execution proceeds.

The first activation record is for the call fact(3). It's created at the beginning of the procedure and holds, among other things, the value of the argument n and the return address where execution should resume after the fact(3) computation is complete.

During the execution of fact(3), we need to make a recursive call to compute fact(2). So that procedure call also gets an activation record with the appropriate values for the argument and return address. Note that the original activation record is kept since it contains information needed to complete the computation of fact(3) after the call to fact(2) returns. So now we have two active procedure calls and hence two activation records.

fact(2) requires computing fact(1), which, in turn, requires computing fact(0). At this point there are four active procedure calls and hence four activation records.

The recursion terminates with fact(0), which returns the value 1 to its caller. At this point we've finished execution of fact(0) and so its activation record is no longer needed and can be discarded.

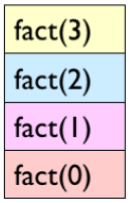
fact(1) now finishes its computation returning 1 to its caller. We no longer need its activation record. Then fact(2) completes, returning 2 to its caller and its activation can be discarded. And so on…

Note that the activation record of a nested procedure call is always discarded before the activation record of the caller. That makes sense: the execution of the caller can't complete until the nested procedure call returns. What we need is a storage scheme that efficiently supports the allocation and deallocation of activation records as shown here.

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

- Stack: push, pop, access to top element



- For C, we only need to access to the activation record of the currently executing procedure

Early compiler writers recognized that activation records are allocated and deallocated in last-in first-out (LIFO) order. So they invented the "stack", a data structure that implements a PUSH operation to add a record to the top of the stack and a POP operation to remove the top element. New activation records are PUSHed onto the stack during procedure calls and the POPed from the stack when the procedure call returns. Note that stack operations affect the top (*i.e.*, most recent) record on the stack.

C procedures only need to access the top activation record on the stack. Other programming languages, *e.g.* Java, support accesses to other active activation records. The stack supports both modes of operation.

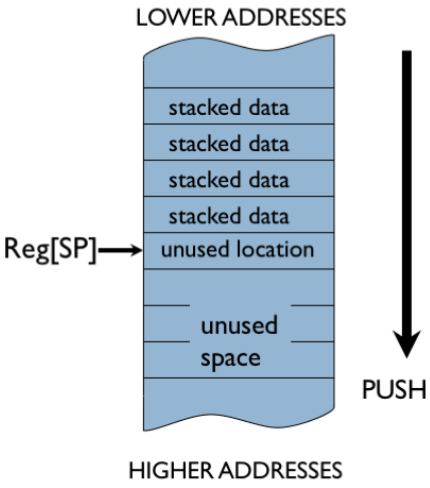
One final technical note: some programming languages support closures (*e.g.*, Javascript) or continuations (*e.g.*, Python's yield statement), where the activation records need to be preserved even after the procedure returns. In these cases, the simple LIFO behavior of the stack is no longer sufficient and we'll need another scheme for allocating and deallocating activation records. But that's a topic for another course!

Stack Implementation

CONVENTIONS:

- Dedicate a register for the Stack Pointer (SP = R29).
- Builds *up* (towards higher addresses) on PUSH
- SP points to first UNUSED location; locations with addresses lower than SP have been previously allocated.
- Discipline: can use stack *at any time*; but leave it as you found it!
- Reserve a large block of memory well away from our program and its data

We use only *software conventions* to implement our stack (many architectures dedicate hardware)



Other possible implementations include stacks that grow "down", SP points to top of stack, etc.

Here's how we'll implement the stack on the Beta:

We'll dedicate one of the Beta registers, R29, to be the "stack pointer" that will be used to manage stack operations.

When we PUSH a word onto the stack, we'll increment the stack pointer. So the stack grows to successively higher addresses as words are PUSHed onto the stack.

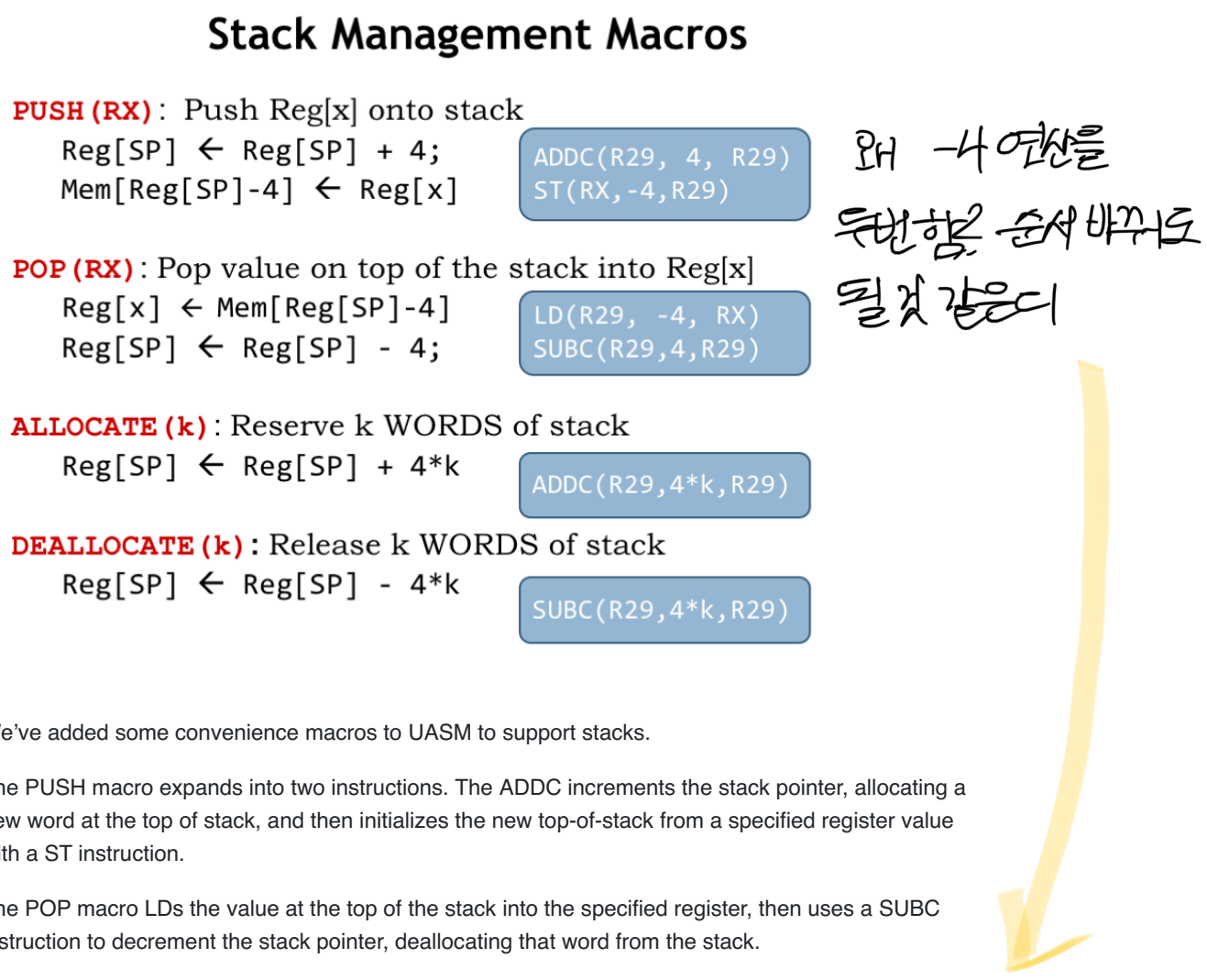
We'll adopt the convention that SP points to (*i.e.*, its value is the address of) the first unused stack location, the location that will be filled by next PUSH. So locations with addresses lower than the value in SP correspond to words that have been previously allocated.

Words can be PUSHed to or POPed from the stack at any point in execution, but we'll impose the rule that code sequences that PUSH words onto the stack must POP those words at the end of execution. So when a code sequence finishes execution, SP will have the same value as it had before the sequence started. This is called the "stack discipline" and ensures that intervening uses of the stack don't affect later stack references.

We'll allocate a large region of memory to hold the stack located so that the stack can grow without overwriting other program storage. Most systems require that you specify a maximum stack size when running a program and will signal an execution error if the program attempts to PUSH too many items onto the stack.

For our Beta stack implementation, we'll use existing instructions to implement stack operations, so for us the stack is strictly a set of software conventions. Other ISAs provide instructions specifically for stack operations.

There are many other sensible stack conventions, so you'll need to read up on the conventions adopted by the particular ISA or programming language you'll be using.



We've added some convenience macros to UASM to support stacks.

The PUSH macro expands into two instructions. The ADDC increments the stack pointer, allocating a new word at the top of stack, and then initializes the new top-of-stack from a specified register value with a ST instruction.

The POP macro LDs the value at the top of the stack into the specified register, then uses a SUBC instruction to decrement the stack pointer, deallocating that word from the stack.

Note that the order of the instructions in the PUSH and POP macro is very important. As we'll see in the next lecture, interrupts can cause the Beta hardware to stop executing the current program between any two instructions, so we have to be careful about the order of operations. So for PUSH, we first allocate the word on the stack, then initialize it. If we did it the other way around and execution was interrupted between the initialization and allocation, code run during the interrupt which uses the stack might unintentionally overwrite the initialized value. But, assuming all code follows stack

discipline, allocation followed by initialization is always safe.

The same reasoning applies to the order of the POP instructions. We first access the top-of-stack one last time to retrieve its value, then we deallocate that location.

We can use the ALLOCATE macro to reserve a number of stack locations for later use. Sort of like PUSH but without the initialization.

DEALLOCATE performs the opposite operation, removing N words from the stack.

In general, if we see a PUSH or ALLOCATE in an assembly language program, we should be able to find the corresponding POP or DEALLOCATE, which would indicate that stack discipline is maintained.

Fun with Stacks

We can use stacks to save values we'll need later. For instance, the following code fragment can be inserted anywhere within a program.

```
          // Argh!!! I'm out of registers Scotty!!
          //
          PUSH(R0)            // Frees up R0
          PUSH(R1)            // Frees up R1
          LD(dilithum_xtals, R0)
          LD(seconds_til_explosion, R1)
suspense: SUBC(R1, 1, R1)
          BNE(R1, suspense)
          ST(R0, warp_engines)
          POP(R1)             // Restores R1
          POP(R0)             // Restores R0
```

Data is popped off the stack in the opposite order that it is pushed on

Next, we'll use show how to use stacks for activation records..

We'll use stacks to save values we'll need later. For example, if we need to use some registers for a computation but don't know if the register's current values are needed later in the program, we can PUSH their current values onto the stack and then are free to use the registers in our code. After we're done, we can use POP to restore the saved values.

Note that we POP data off the stack in the opposite order that the data was PUSHed, *i.e.*, we need to follow the last-in first-out discipline imposed by the stack operations.

Now that we have the stack data structure, we'll use it to solve our problems with allocating and deallocating activation records during procedure calls.

Solving Procedure Linkage Problems

Reminder: Procedure storage needs

1) We need a way to *pass arguments* to the procedure
2) Procedures need their own *LOCAL storage*
3) Procedures need to *call other procedures*; special case: recursive procedures that *call themselves*

Plan for caller:

- Push argument values

C code:

$proc(expr_1, \ldots, expr_n)$

onto stack *in reverse order* for use by callee

- Branch to callee, save return address in dedicated register (LP = R28)
- Clean up stack after callee return

Beta assembly:

```
compile_expr(expr_n)⇒Rx
PUSH(rx)
...
compile_expr(expr_1)⇒Rx
PUSH(rx)
BR(proc,LP)
DEALLOCATE(n)
```

We'll use the stack to hold a procedure's activation record. That includes the values of the arguments to the procedure call. We'll allocate words on the stack to hold the values of the procedure's local variables, assuming we don't keep them in registers. And we'll use the stack to save the return address (passed in LP) so the procedure can make nested procedure calls without overwriting its return address.

The responsibility for allocating and deallocating the activation record will be shared between the calling procedure (the "caller") and the called procedure (the "callee").

The caller is responsible for evaluating the argument expressions and saving their values in the activation record being built on the stack. We'll adopt the convention that argument values are pushed in reverse order, *i.e.*, the first argument will be the last to be pushed on the stack. We'll explain why we made this choice in a couple of slides…

The code compiled for a procedure involves a sequence of expression evaluations, each followed by a PUSH to save the calculated value on the stack. So when the callee starts execution, the top of the stack contains the value of the first argument, the next word down the value of the second argument, and so on.

After the argument values, if any, have been pushed on the stack, there's a BR to transfer control to the procedure's entry point, saving the address of the instruction following the BR in the linkage pointer, R28, a register that we'll dedicate to that task.

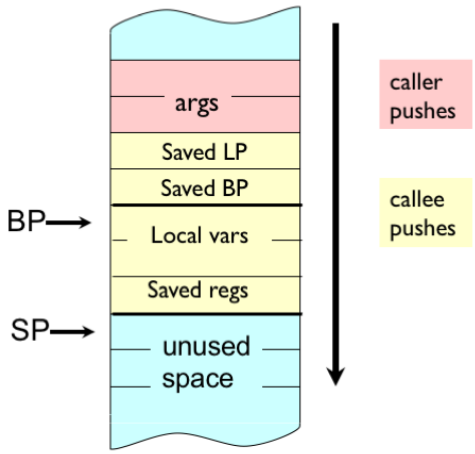
When the callee returns and execution resumes in the caller, a DEALLOCATE is used to remove all the argument values from the stack, preserving stack discipline.

So that's the code the compiler generates for the procedure. The rest of the work happens in the called procedure.

Stack Frames as Activation Records

CALLEE uses stack for all of the its storage needs:

1. Saving return address back to the caller (it's in LP)
2. Saving BP of caller (pointer to caller's activation record)
3. Allocating stack locations to hold local variables
4. Save any registers callee uses: "callee saves" convention



Dedicate another register (BP = R27) to hold address of the activation record. Use when accessing

- Arguments
- Other local storage

BP is a convenience

In theory it's possible to use SP to access stack frame, but offsets will change due to PUSHs and POPs. For convenience we use BP so we can use constant offsets to find, e.g., the first argument.

The code at the start of the called procedure completes the allocation of the activation record. Since when we're done the activation record will occupy a bunch of consecutive words on the stack, we'll sometimes refer to the activation record as a "stack frame" to remind us of where it lives.

The first action is to save the return address found in the LP register. This frees up LP to be used by any nested procedure calls in the body of the callee.

In order to make it easy to access values stored in the activation record, we'll dedicate another register called the "base pointer" (BP = R27) which will point to the stack frame we're building. So as we enter the procedure, the code saves the pointer to the caller's stack frame, and then uses the current value of the stack pointer to make BP point to the current stack frame. We'll see how we use BP in just a moment. 4를 더하고 저장하니까 가능?

Now the code will allocate words in the stack frame to hold the values for the callee's local variables, if any.

Finally, the callee needs to save the values of any registers it will use when executing the rest of its code. These saved values can be used to restore the register values just before returning to the caller. This is called the "callee saves" convention where the callee guarantees that all register values will be preserved across the procedure call. With this convention, the code in the caller can assume any values it placed in registers before a nested procedure call will still be there when the nested call returns.

Note that dedicating a register as the base pointer isn't strictly necessary. All accesses to the values on the stack can be made relative to the stack pointer, but the offsets from SP will change as values are PUSHed and POPed from the stack, *e.g.*, during procedure calls. It will be easier to understand the generated code if we use BP for all stack frame references.

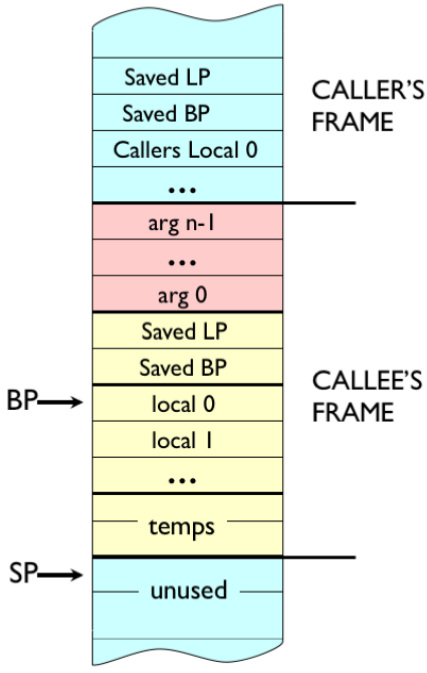
Stack Frame Details

CALLER passes arguments to CALLEE on the stack in *reverse* order

F(1,2,3,4) translates to:

```
CMOVE(4,R0)
PUSH(R0)
CMOVE(3,R0)
PUSH(R0)
CMOVE(2,R0)
PUSH(R0)
CMOVE(1,R0)
PUSH(R0)
BR(F, LP)
```

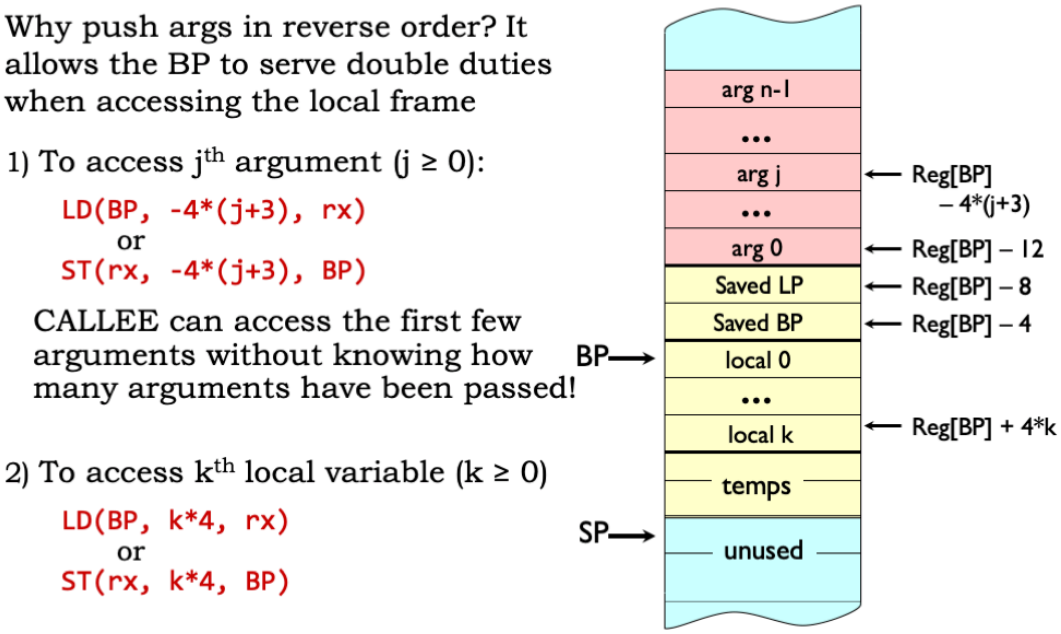
Why push args in REVERSE order?



Let's return to the question about the order of argument values in the stack frame. We adopted the convention of PUSHing the values in reverse order, *i.e.*, where the value of the first argument is the last one to be PUSHED.

So, why PUSH argument values in reverse order?

Argument Order & BP Usage



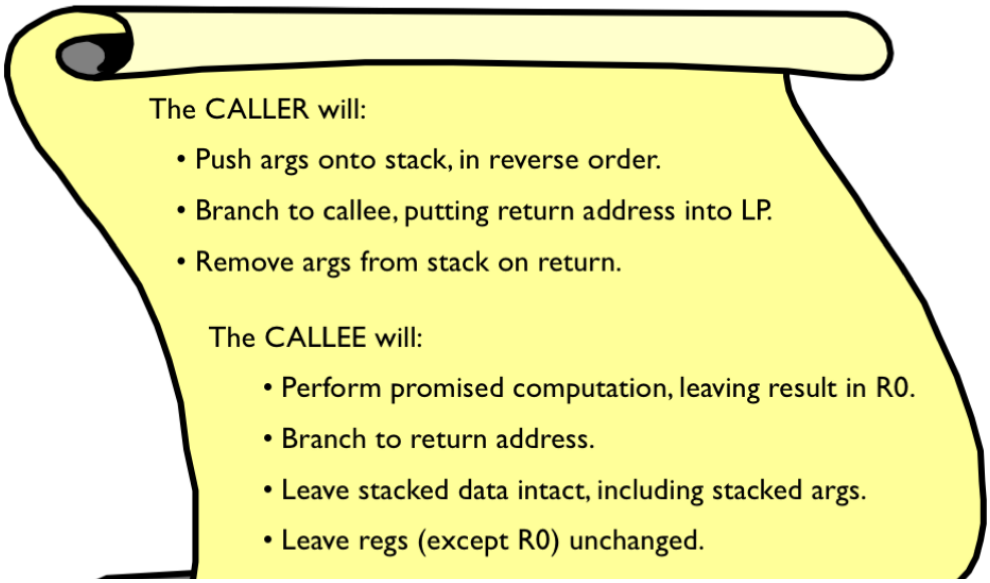
With the arguments PUSHed in reverse order, the first argument (labeled “arg 0”) will be at a fixed offset from the base pointer regardless of the number of argument values pushed on the stack. The compiler can use a simple formula to the determine the correct BP offset value for any particular argument. So the first argument is at offset -12, the second at -16, and so on.

Why is this important? Some languages, such as C, support procedure calls with a variable number of arguments. Usually the procedure can determine from, say, the first argument, how many additional arguments to expect. The canonical example is the C printf function where the first argument is a format string that specifies how a sequence of values should be printed. So a call to printf includes the format string argument plus a varying number of additional arguments. With our calling convention the format string will always be in the same location relative to BP, so the printf code can find it without knowing the number of additional arguments in the current call.

The local variables are also at fixed offsets from BP. The first local variable is at offset 0, the second at offset 4, and so on.

So we see that having a base pointer makes it easy to access the values of the arguments and local variables using fixed offsets that can be determined at compile time. The stack above the local variables is available for other uses, *e.g.*, building the activation record for a nested procedure call!

Procedure Linkage: The Contract



Okay, here's our final contract for how procedure calls will work:

The calling procedure ("caller") will

PUSH the argument values onto the stack in reverse order.

Branch to the entry point of the callee, putting the return address into the linkage pointer.

When the callee returns, remove the argument values from the stack.

The called procedure ("callee") will

Perform the promised computation, leaving the result in R0.

Jump to the return address when the computation has finished.

Remove any items it has placed on the stack, leaving the stack as it was when the procedure was entered. Note that the arguments were PUSHed on the stack by the caller, so it will be up to the caller to remove them.

Preserve the values in all registers except R0, which holds the return value. So the caller can assume any values placed in registers before a nested call will be there when the nested call returns.

Procedure Linkage Templates

Calling Sequence

```
    PUSH(arg_n)                 // push args, last arg first
    ...
    PUSH(arg_1)
    BR(f, LP)                   // Call f.
    DEALLOCATE(n)               // Clean up!
    ...                         // (f's return value in r0)
```

Entry Sequence

```
f:  PUSH(LP)                    // Save LP and BP
    PUSH(BP)                    // in case we make new calls.
    MOVE(SP,BP)                 // set BP=frame base
    ALLOCATE(nlocals)           // allocate locals
    (push other regs)           // preserve any regs used
```

Exit Sequence

```
    // return value in R0…
    (pop other regs)            // restore regs
    MOVE(BP,SP)                 // strip locals, etc
    POP(BP)                     // restore CALLER's linkage
    POP(LP)                     // (the return address)
    JMP(LP)                     // return.
```

Why no DEALLOCATE?

We saw the code template for procedure calls on an earlier slide.

Here's the template for the entry point to a procedure F. The code saves the caller's LP and BP values, initializes BP for the current stack frame and allocates words on the stack to hold any local variable values. The final step is to PUSH the value of any registers (besides R0) that will be used by the remainder of the procedure's code.

The template for the exit sequence mirrors the actions of the entry sequence, restoring all the values saved by the entry sequence, performing the POP operations in the reverse of the order of the PUSH operations in the entry sequence. Note that in moving the BP value into SP we've reset the stack to its state at the point of the MOVE(SP,BP) in the entry sequence. This implicitly undoes the effect of the ALLOCATE statement in the entry sequence, so we don't need a matching DEALLOCATE in the exit sequence.

The last instruction in the exit sequence transfers control back to the calling procedure.

With practice you'll become familiar with these code templates. Meanwhile, you can refer back to this slide whenever you need to generate code for a procedure call.

Putting It All Together: Factorial

```
int fact(int n) {
  if (n > 0) {
    return n*fact(n-1);
  } else {
    return 1;
  }
}
```

```
fact:   PUSH(LP)          // save linkages
        PUSH(BP)
        MOVE(SP,BP)       // new frame base
        PUSH(r1)          // preserve regs
        LD(BP,-12,r1)     // r1 ← n
        CMPLEC(r1,0,r0)   // if (n > 0)
        BT(r0,else)

        SUBC(r1,1,r1)     // r1 ← (n-1)
        PUSH(r1)          // push arg1
        BR(fact,LP)       // fact(n-1)
        DEALLOCATE(1)     // pop arg1
        LD(BP,-12,r1)     // r1 ← n
        MUL(r1,r0,r0)     // r0 ← n*fact(n-1)
        BR(rtn)

else:   CMOVE(1,r0)       // return 1

rtn:    POP(r1)           // restore regs
        MOVE(BP,SP)       // Why?
        POP(BP)           // restore links
        POP(LP)
        JMP(LP)           // return
```

Here's the code our compiler would generate for the C implementation of factorial shown on the left.

The entry sequence saves the caller's LP and BP, then initializes BP for the current stack frame. The value of R1 is saved so we can use R1 in code that follows.

The exit sequence restores all the saved values, including that for R1. The code for the body of the procedure has arranged for R0 to contain the return value by the time execution reaches the exit sequence.

The nested procedure call passes the argument value on the stack and removes it after the nested call returns.

The remainder of the code is generated using the templates we saw in the previous lecture. Aside from computing and pushing the values of the arguments, there are approximately 10 instructions needed to implement the linking approach to a procedure call. That's not much for a procedure of any size, but might be significant for a trivial procedure. As mentioned earlier, some optimizing compilers can make the tradeoff of inlining small non-recursive procedures saving this small amount of overhead.

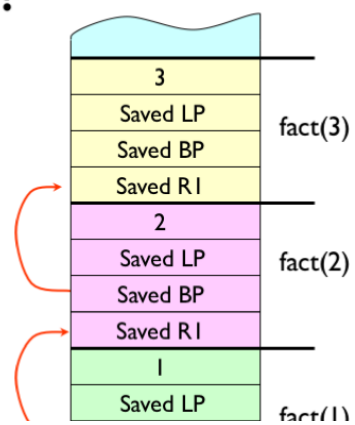
Recursion?

Of course!

- Frames allocated for each recursive call…
- Deallocated (in inverse order) as recursive calls return

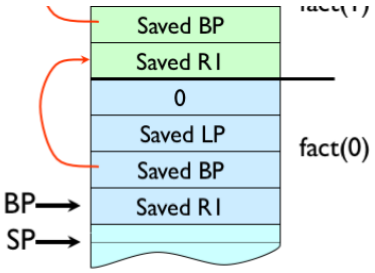
Debugging skill:
"stack crawling"

- Given code, stack snapshot –



- Given code, stack snapshot – figure out what, where, how, who...
- Follow old <BP> links to parse frames
- Decode args, locals, return locations, etc etc etc

Particularly useful on 6.004 quizzes!



So have we solved the activation record storage issue for recursive procedures?

Yes! A new stack frame is allocated for each procedure call. In each frame we see the storage for the argument and return address [illegible]. And as the nested calls return the stack frames will be deallocated in inverse order.

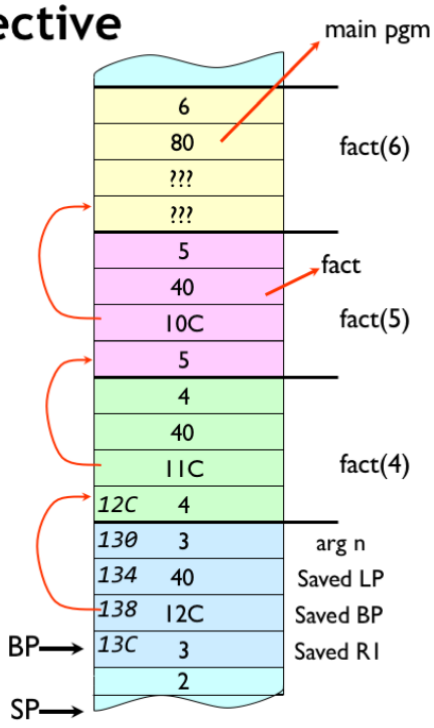
Interestingly we can learn a lot about the current state of execution by looking at the active stack frames. The current value of BP, along with the older values saved in the activation records, allow us to identify the active procedure calls and determine their arguments, the values of any local variables for active calls, and so on. If we print out all this information at any given time we would have a "stack trace" showing the progress of the computation. In fact, when a problem occurs, many language runtimes will print out the stack trace to help the programmer determine what happened.

And, of course, if you can interpret the information in the stack frames, you can show you understand our conventions for procedure call and return. Don't be surprised to find such a question on a quiz :)

Stack Detective

`fact(n)` is called. During the calculation, the computer is stopped with the PC at 0x40; the stack contents are shown (in hex).

- What's the argument to the *active* call to fact? 3
- What's the argument to the *original* call to fact? 6
- What's the location of the original calling (BR) instruction? 80 – 4 = 7C
- What instruction is about to be executed? DEALLOCATE(1)
- What value is in BP? 13C
- What value is in SP? 13C+4+4=144
- What value is in R0? fact(2) = 2



Let's practice our newfound skill and see what we can determine about a running program which we've stopped somewhere in the middle of its execution. We're told that a computation of fact() is in progress and that the PC of the next instruction to be executed is 0x40. We're also given the stack dump shown on right.

Since we're in the middle of a fact computation, we know that current stack frame (and possibly others) is an activation record for the fact function. Using the code on the previous slide we can determine the layout of the stack frame and generate the annotations shown on the right of the stack dump. With the annotations, it's easy to see that the argument to current active call is the value 3.

Now we want to know the argument to original call to fact. We'll have to label the other stack frames using the saved BP values. Looking at the saved LP values for each frame (always found at an offset of -8 from the frame's BP), we see that many of the saved values are 0x40, which must be the return

of -3 from the frame's BP), we see that many of the saved values are 0x40, which must be the return address for the recursive fact calls.

Looking through the stack frames we find the first return address that's *not* 0x40, which must an return address to code that's not part of the fact procedure. This means we've found the stack frame created by the original call to fact and can see that argument to the original call is 6.

What's the location of the BR that made the original call? Well the saved LP in the stack frame of the original call to fact is 0x80. That's the address of the instruction following the original call, so the BR that made the original call is one instruction earlier, at location 0x7C. To answer these questions you have to be good at hex arithmetic!

What instruction is about to be executed? We were told its address is 0x40, which we notice is the saved LP value for all the recursive fact calls. So 0x40 must be the address of the instruction following the BR(fact,LP) instruction in the fact code. Looking back a few slides at the fact code, we see that's a DEALLOCATE(1) instruction.

What value is in BP? Hmm. We know BP is the address of the stack location containing the saved R1 value in the current stack frame. So the saved BP value in the current stack frame is the address of the saved R1 value in the *previous* stack frame. So the saved BP value gives us the address of a particular stack location, from which we can derive the address of all the other locations! Counting forward, we find that the value in BP must be 0x13C.

What value is in SP? Since we're about to execute the DEALLOCATE to remove the argument of the nested call from the stack, that argument must still be on the stack right after the saved R1 value. Since the SP points to first unused stack location, it points to the location after that word, so it has the value 0x144.

Finally, what value is is R0? Since we've just returned from a call to fact(2) the value in R0 must the result from that recursive call, which is 2.

Wow! You can learn a lot from the stacked activation records and a little deduction! Since the state of the computation is represented by the values of the PC, the registers, and main memory, once we're given that information we can tell exactly what the program has been up to. Pretty neat…

Summary of Dedicated Registers

The Beta ISA has 32 registers. But we've dedicated several of them to serve a specific purpose:

- R31 is always zero [ISA]
- R30 … reserved for future use… [next lecture]
- R29 = SP, stack pointer [software convention]
- R28 = LP, linkage pointer [software convention]
- R27 = BP, base pointer [software convention]

Wrapping up, we've been dedicating some registers to help with our various software conventions. To summarize:

R31 is always zero, as defined by the ISA.

We'll also dedicate R30 to a particular function in the ISA when we discuss the implementation of the Beta in the next lecture. Meanwhile, don't use R30 in your code!

The remaining dedicated registers are connected with our software conventions:

R29 (SP) is used as the stack pointer,

R28 (LP) is used as the linkage pointer, and

R27 (BP) is used as the base pointer.

As you practice reading and writing code, you'll grow familiar with these dedicated registers.

Summary

- Each procedure invocation has an activation record
  - Created during procedure call/entry sequence
  - Discarded when procedure returns
  - Holds:
    - Argument values (in reverse order)
    - Saved LP, BP from caller (callee reuses those regs)
    - Storage for local variables (if any)
    - Other saved regs from caller (callee needs regs to use)
  - BP points to activation record of active call
    - Access arguments at offsets of -12, -16, -20, ..
    - Access local variables at offsets of 0, 4, 8, …
- "Callee saves" convention: all reg values preserved
- Except for R0, which holds return value

In thinking about how to implement procedures, we discovered the need for an activation record to store the information needed by any active procedure call.

An activation record is created by the caller and callee at the start of a procedure call. And the record can be discarded when the procedure is complete.

The activation records hold argument values, saved LP and BP values along with the caller's values in any other of the registers. Storage for the procedure's local variables is also allocated in the activation record.

We use BP to point to the current activation record, giving easy access to the values of the arguments and local variables.

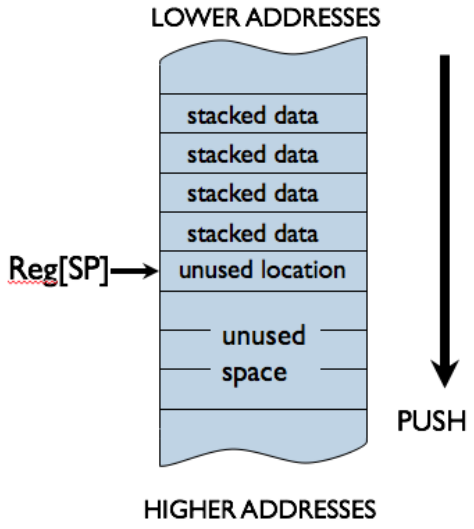
We adopted a "callee saves" convention where the called procedure is obligated to preserve the values in all registers except for R0.

Taken together, these conventions allow us to have procedures with arbitrary numbers of arguments and local variables, with nested and recursive procedure calls. We're now ready to compile and execute any C program!

- BackProcedures and Stacks
- ContinueTopic Videos

Computation Structures

Procedures & Stacks Worksheet



PUSH(X): Push Reg[x] onto stack

```
ADDC(SP,4,SP)
ST(Rx,-4,SP)
```

POP(X): Pop value at top of stack into Reg[x]

```
LD(SP,-4,RX)
SUBC(SP,4,SP)
```

ALLOCATE(k): Reserve k words of stack

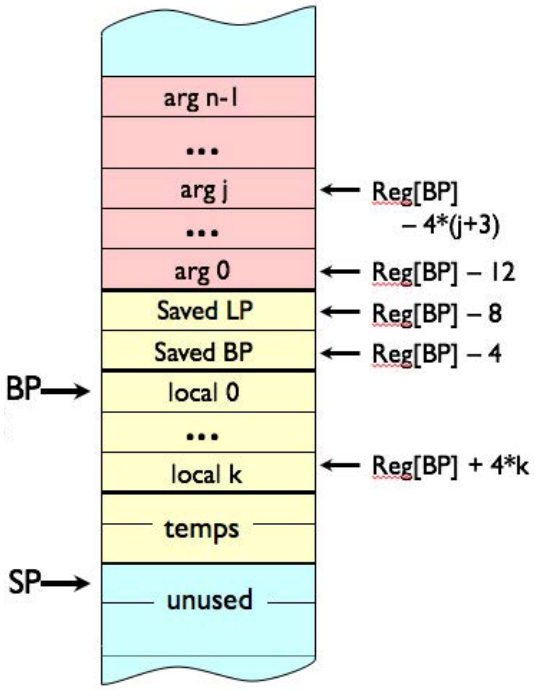
```
ADDC(SP,4*k,SP)
```

DEALLOCATE(k): Release k words of stack

```
SUBC(SP,4*k,SP)
```

Stack discipline: leave stack the way you found it => for every PUSH(), there's a corresponding POP() or DEALLOCATE()

Activation record layout on the stack (aka stack frame):



CALLING SEQUENCE

```
   PUSH(argn)    // push args, last arg first
   …
   PUSH(arg1)
   BR(f, LP)     // call f, return addr in LP
   DEALLOCATE(n) // remove args from stack
```

ENTRY SEQUENCE

```
f: PUSH(LP)      // save return addr
   PUSH(BP)      // save old frame pointer
   MOVE(SP,BP)   // initialize new frame pointer
   ALLOCATE(nlocals)  // make room for locals
   (push other regs)  // preserve old reg vals
```

EXIT SEQUENCE

```
   // return value in R0
   MOVE(BP,SP)   // remove locals
   POP(BP)       // restore old frame pointer
   POP(LP)       // recover return address
   JMP(LP)       // resume execution in caller
```

Problem 1.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int fn(int x) {
  int lowbit = x & 1;
  int rest = x >> 1;
  if (x == 0) return 0;
  else return ???;
}
```

```
fn: PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(2)
    PUSH(R1)
    LD(BP,-12,R0)
    ANDC(R0,1,R1)
xx: ST(R1,0,BP)
    SHRC(R0,1,R1)
    ST(R1,4,BP)
yy: BEQ(R0,rtn)
    LD(BP,4,R1)
    PUSH(R1)
    BR(fn,LP)
    DEALLOCATE(1)
    LD(BP,0,R1)
    ADD(R1,R0,R0)
rtn:POP(R1)
zz: MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: lowbit + fn(lowbit)

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure (circle one)?

Work the same? YES … (NO)

(C) In the space below, fill in the binary representation for the instruction stored at the location tagged '**xx:**' in the above program.

011001	11001	00001	00000000000000000

(fill in missing 1s and 0s for instruction at xx:)

The procedure **fn** is called from an external procedure and its execution is interrupted just prior to the execution of the instruction tagged '**yy:**'. The contents of a region of memory are shown on the left below.

NB: All addresses and data values are shown in hex. The contents of **BP** are 0x1C8 and **SP** contains 0x1D4.

```
184:   4
188:   7
18C:  47
190:  C4  LP
194: 170  BP
198:   1
19C:  23
1A0:  22
1A4:  23  X
1A8:  4C  LP
1AC: 198  BP
1B0:   1
1B4:  11
1B8:  23
1BC:  11  X
1C0:  4C  LP
1C4: 1B0  BP
1C8:   1  ←BP
1CC:   8
1D0: ???  R1
1D4:   0  ←SP
```

(D) What was the argument to the most recent call to **fn**?

Most recent argument (HEX): x= 0x11

(E) What is the missing value marked **???** for the contents of location 1D0?

R1값. 이전 BP+4

Contents of 1D0 (HEX): 0x11

(F) What is the hex address of the instruction tagged **rtn:?**

Address of rtn (HEX): 0x58

(G) What was the argument to the *original* call to **fn**?

Original argument (HEX): x= 0x47

(H) What is the hex address of the BR instruction that called **fn** *originally*?

Address of original call (HEX): 0x90

(I) What were the contents of R1 at the time of the *original* call?

Original R1 contents (HEX): 0x22

(J) What value will be returned to the *original* caller?

Return value for original call (HEX): 0x4

1000 0111

Problem 2.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int f(int x, int y) {
  x = (x >> 1) + y;
  if (y == 0) return x;
  else return ???;
}
```

```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP,BP)
      PUSH(R1)
      LD(BP,-12,R0)
      SHRC(R0,1,R0)
      LD(BP,-16,R1)
      ADD(R0,R1,R0)
      BEQ(R1,rtn)
      SUBC(R1,1,R1)
      PUSH(R1)
      PUSH(R0)
      BR(f,LP)
      DEALLOCATE(2)
rtn:  POP(R1)
zz:   MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: f(x,y-1)

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure?

Work the same (circle one)? YES … NO

The procedure **f** is called from an external procedure and then execution is stopped just prior to one of the executions of the instruction labeled '**rtn:**'. The addresses and contents of a region of memory are shown in the table on the right; all addresses and data values in the table are in hex. When execution is stopped **BP contains the value 0x14C and SP contains the value 0x150.**

108	7
10C	320
110	104
114	3
118	A
11C	2C4
120	104
124	3 BP
128	2
12C	
130	348
134	124
138	2 BP
13C	1
140	6
144	348
148	138
14C	1 BP
150	0 SP
154	4
158	348
15C	14C
160	0

(C) What are the arguments to the **currently active call** to **f**?

Most recent arguments (in hex): x = 0x 6 **, y = 0x** 1

(D) If you can tell from the information provided, specify the arguments to the **original** call to **f**, otherwise select **CAN'T TELL**.

Original arguments (in hex) : x = 0x A **, y = 0x** 3 **, or CAN'T TELL**

(E) What is the missing value in location 0x12C?

Contents of location 0x12C (in hex): 0x 8

(F) What is the hex address of the instruction labeled **rtn:?**

Address of instruction labeled rtn: (in hex): 0x 15C

(G) What is the hex address of the BR instruction that called **f** *originally*?

Address of original call (in hex): 0x 200 **, or CAN'T TELL**

(H) What value will be returned to the *original* caller?

Return value for original call (in hex): 0x 2

Problem 3.

The following C program implements a function H(x,y) of two arguments, which returns an integer result. The assembly code for the procedure is shown on the right.

```
int H(int x, int y) {
    int a = x - y;
    if (a < 0) return x;
    else return ???;
}
```

```
H:  PUSH(LP)
    PUSH(BP)
    MOVE(SP, BP)
    ALLOCATE(1)
    PUSH(R1)

    LD(BP,-12,R0)
    LD(BP,-16,R1)
    SUB(R0,R1,R1)
    ST(R1,0,BP)

    CMPLTC(R1,0,R1)
    BT(R1,rtn)

    LD(BP,-16,R1)
    PUSH(R1)
    LD(BP,0,R0)
    PUSH(R0)
    BR(H,LP)
    DEALLOCATE(2)
rtn: POP(R1)
    MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP)
```

The execution of the procedure call **H(0x68,0x20)** has been suspended just as the Beta is about to execute the instruction labeled "rtn:" during one of the recursive calls to H. A *partial* trace of the stack at the time execution was suspended is shown to the right below.

(A) Examining the assembly language for H, what is the appropriate C code for ??? in the C representation for H?

C code for ???: _____

(B) Please fill in the values for the blank locations in the stack dump shown on the right. Express the values in hex or write "---" if value can't be determined. Hint: Figure out the layout of H's activation record and use it to identify and label the stack frames in the stack dump.

Fill in the blank locations with values (in hex!) or "---"

(C) Determine the specified values at the time execution was suspended. Please express each value in hex or write "CAN'T TELL" if the value cannot be determined.

Value in R0 or "CANT TELL": 0x_____

Value in R1 or "CANT TELL": 0x_____

Value in BP or "CANT TELL": 0x_____

Value in LP or "CANT TELL": 0x_____

Value in SP or "CANT TELL": 0x_____

	0x0024
	0x0070
	0x0048
	0x0068
	0x0020
	0x0020
	0x0028
	0x007C
	0x00C8
BP→	0x0008
	0x0020
	0x0020

Problem 4.

The following C program computes the log base 2 of its argument. The assembly code for the procedure is shown on the right, along with a stack trace showing the execution of ilog2(10). The execution has been halted just as it's about to execute the instruction labeled "rtn:"

```
/* compute log base 2 of arg */
int ilog2(unsigned x) {
    unsigned y;
    if (x == 0) return 0;
    else {
        /* shift x right by 1 bit */
        y = x >> 1;
        return ilog2(y) + 1;
    }
}
```

```
ilog2: PUSH(LP)
       PUSH(BP)
       MOVE(SP,BP)
       ALLOCATE(1)
       PUSH(R1)

       LD(BP,-12,R0)
       BEQ(R0,rtn,R31)

       LD(BP,-12,R1)
       SHRC(R1,1,R1)
       ST(R1,0,BP)

       LD(BP,0,R1)
       PUSH(R1)
       BR(ilog2,LP)
       DEALLOCATE(1)
       ADDC(R0,1,R0)

rtn:   POP(R1)
xxx:   DEALLOCATE(1)
       MOVE(BP,SP)
       POP(BP)
       POP(LP)
       JMP(LP)
```

(A) What are the values in R0, SP, BP and LP at the time execution was halted? Please express the values in hex or write "CAN'T TELL".

Value in R0: 0x 1 **in SP: 0x** 24C

Value in BP: 0x 244 **in LP: 0x** 1A8

(B) Please fill in the values for the five blank locations in the stack trace shown on the right. Please express the values in hex.

Fill in values (in hex!) for 5 blank locations

	5
	1A8
	208
	2
	5
	2
	1A8
	21C
	1
	2
	1
	1A8
	230
BP→	0
	1
	0

Values are in hex!

(C) In the assembly language code for ilog2 there is the instruction "LD(BP,-12,R0)". If this instruction were rewritten as "LD(SP,NNN,R0)" what is correct value to use for NNN?

Correct value for NNN: —

(D) In the assembly language code for ilog2, what is the address of the memory location labeled "xxx:"? Please express the value in hex.

Address of location labeled "xxx:": 0x 1B

Problem 1.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int fn(int x) {
  int lowbit = x & 1;
  int rest = x >> 1;
  if (x == 0) return 0;
  else return ???;
}
```

```
fn: PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(2)
    PUSH(R1)
    LD(BP,-12,R0)
    ANDC(R0,1,R1)
xx: ST(R1,0,BP)
    SHRC(R0,1,R1)
    ST(R1,4,BP)
yy: BEQ(R0,rtn)
    LD(BP,4,R1)
    PUSH(R1)
    BR(fn,LP)
    DEALLOCATE(1)
    LD(BP,0,R1)
    ADD(R1,R0,R0)
rtn:POP(R1)
zz: MOVE(BP,SP)
    POP(BP)
    POP(LP)
    JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: fn(rest) + lowbit

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure (circle one)?

Work the same? YES ... (NO)

The MOVE(BP,SP) is deallocating the local variables lowbit and rest

(C) In the space below, fill in the binary representation for the instruction stored at the location tagged '**xx:**' in the above program.

R27

0 1 1 0 0 1	0 0 0 0 1	1 1 0 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ST	R1	BP	0

(fill in missing 1s and 0s for instruction at xx:)

The procedure **fn** is called from an external procedure and its execution is interrupted just prior to the execution of the instruction tagged '**yy:**'. The contents of a region of memory are shown on the left below.

NB: All addresses and data values are shown in hex. The contents of **BP** are 0x1C8 and **SP** contains 0x1D4.

(D) What was the argument to the most recent call to **fn**?

```
184:   4
188:   7
18C:  47   X
190:  C4   saved LP
194: 170   saved BP
198:   1   lowbit
19C:  23   rest
1A0:  22   saved R1
1A4:  23   X
1A8:  4C   saved LP
1AC: 198   saved BP
1B0:   1   lowbit
1B4:  11   rest
1B8:  23   saved R1
1BC:  11   X
1C0:  4C   saved LP
1C4: 1B0   saved BP
1C8:   1   ←BP lowbit
1CC:   8   rest
1D0: ???   saved R1
1D4:   0   ←SP
```

from current stack frame **Most recent argument (HEX): x=** 11

(E) What is the missing value marked **???** for the contents of location 1D0?

looking at code: R1 held value of argument before the call

Contents of 1D0 (HEX): 11

(F) What is the hex address of the instruction tagged **rtn:?**

saved LP is the address of mem location holding DEALLOCATE inst.

Address of rtn (HEX): 58

(G) What was the argument to the *original* call to **fn**?

find frame with saved LP ≠ 0x4C.

Original argument (HEX): x= ∃47

(H) What is the hex address of the BR instruction that called **fn** *originally*?

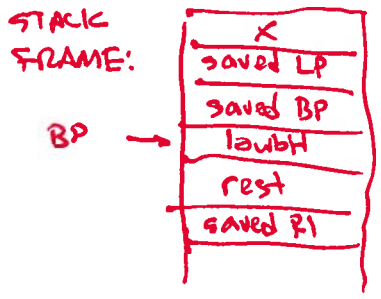
(saved LP on oldest stack frame) − 4.

Address of original call (HEX): CØ

(I) What were the contents of R1 at the time of the *original* call?

Original R1 contents (HEX): 22

(J) What value will be returned to the *original* caller?



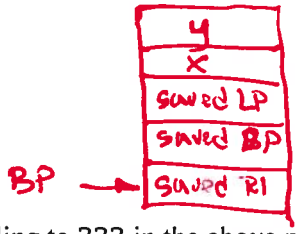
Return value for original call (HEX): 4

fn counts number of 1 bits in argument.

Problem 2.

You are given an incomplete listing of a C program (shown below) and its translation to Beta assembly code (shown on the right):

```
int f(int x, int y) {
  x = (x >> 1) + y;
  if (y == 0) return x;
  else return ???;
}
```




```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP,BP)
      PUSH(R1)
      LD(BP,-12,R0)
      SHRC(R0,1,R0)
      LD(BP,-16,R1)
      ADD(R0,R1,R0)
      BEQ(R1,rtn)
      SUBC(R1,1,R1)
      PUSH(R1)
      PUSH(R0)
      BR(f,LP)
      DEALLOCATE(2)
rtn:  POP(R1)
zz:   MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: f(x, y-1)

(B) Suppose the instruction bearing the tag '**zz:**' were eliminated from the assembly language program. Would the modified procedure work the same as the original procedure?

Work the same (circle one)? (YES) ... NO

The procedure **f** is called from an external procedure and then execution is stopped just prior to one of the executions of the instruction labeled '**rtn:**'. The addresses and contents of a region of memory are shown in the table on the right; all addresses and data values in the table are in hex. When execution is stopped **BP contains the value 0x14C and SP contains the value 0x150.**

(C) What are the arguments to the **currently active call** to **f**?

Most recent arguments (in hex): x = 0x 6 , y = 0x 1

(D) If you can tell from the information provided, specify the arguments to the **original** call to **f**, otherwise select **CAN'T TELL.**

Original arguments (in hex) : x = 0x A , y = 0x 3 , **or CAN'T TELL**

(E) What is the missing value in location 0x12C?

x = (0xA >> 1) + y
  = 5   3

Contents of location 0x12C (in hex): 0x 8

(F) What is the hex address of the instruction labeled **rtn:**?

Address of instruction labeled rtn: (in hex): 0x 34C

(G) What is the hex address of the BR instruction that called **f** *originally*?

Address of original call (in hex): 0x 2C0 , **or CAN'T TELL**

(H) What value will be returned to the *original* caller?

Return value for original call (in hex): 0x 2

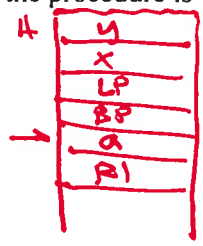
f(A,3) → f(8,2) → f(6,1) → f(9,0) → returns 2

108	7	
10C	320	
110	104	
114	3	y
118	A	x
11C	2C4	LP
120	104	BP
124	3	R1
128	2	y
12C	8	x
130	348	LP
134	124	BP
138	2	R1
13C	1	y
140	6	x
144	348	LP
148	138	BP
BP→14C	1	R1
SP→150	0	
154	4	
158	348	
15C	14C	
160	0	

Problem 3.

The following C program implements a function H(x,y) of two arguments, which returns an integer result. The assembly code for the procedure is shown on the right.

```
int H(int x, int y) {
    int a = x - y;
    if (a < 0) return x;
    else return ???;
}
```




```
H:    PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      ALLOCATE(1)
      PUSH(R1)

      LD(BP,-12,R0)
      LD(BP,-16,R1)
      SUB(R0,R1,R1)
      ST(R1,0,BP)

      CMPLTC(R1,0,R1)
      BT(R1,rtn)

      LD(BP,-16,R1)
      PUSH(R1)
      LD(BP,0,R0)
      PUSH(R0)
      BR(H,LP)
      DEALLOCATE(2)

rtn:  POP(R1)
      MOVE(BP,SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

The execution of the procedure call **H(0x68,0x20)** has been suspended just as the Beta is about to execute the instruction labeled "rtn:" during one of the recursive calls to H. A *partial* trace of the stack at the time execution was suspended is shown to the right below.

(A) Examining the assembly language for H, what is the appropriate C code for ??? in the C representation for H?

C code for ???: H(a,y)

(B) Please fill in the values for the blank locations in the stack dump shown on the right. Express the values in hex or write "---" if value can't be determined. Hint: Figure out the layout of H's activation record and use it to identify and label the stack frames in the stack dump.

Fill in the blank locations with values (in hex!) or "---"

	0x0024	LP	
	0x0070	BP	
	0x0048	a	
	0x0068	R1	
	0x20	y	
	0x68	x	
	0x7C	LP	
	0x80	BP	
C8	0x28	a	
CC	0x0020	R1	
D0	0x0020	y	
D4	0x0028	x	
D8	0x007C	LP	
DC	0x00C8	BP	
E0 BP→	0x0008	a	
E4	0x0020	R1	
SP→ E8	0x0020		

(C) Determine the specified values at the time execution was suspended. Please express each value in hex or write "CAN'T TELL" if the value cannot be determined.

result of H(8,20) **Value in R0 or "CANT TELL": 0x** 8

y **Value in R1 or "CANT TELL": 0x** 20

Value in BP or "CANT TELL": 0x E0

Value in LP or "CANT TELL": 0x 7C

Value in SP or "CANT TELL": 0x E8

Problem 4.

The following C program computes the log base 2 of its argument. The assembly code for the procedure is shown on the right, along with a stack trace showing the execution of ilog2(10). The execution has been halted just as it's about to execute the instruction labeled "rtn:"

```
/* compute log base 2 of arg */
int ilog2(unsigned x) {
    unsigned y;
    if (x == 0) return 0;
    else {
        /* shift x right by 1 bit */
        y = x >> 1;
        return ilog2(y) + 1;
    }
}
```


```
ilog2: PUSH(LP)
       PUSH(BP)
       MOVE(SP,BP)
       ALLOCATE(1)
       PUSH(R1)

       LD(BP,-12,R0)
       BEQ(R0,rtn,R31)

       LD(BP,-12,R1)
       SHRC(R1,1,R1)
       ST(R1,0,BP)

       LD(BP,0,R1)
       PUSH(R1)
       BR(ilog2,LP)
1A8    DEALLOCATE(1)
1AC    ADDC(R0,1,R0)
1B0,1B4
rtn:   POP(R1)          <-- STOP
1B8 xxx: DEALLOCATE(1)
       MOVE(BP,SP)
       POP(BP)
       POP(LP)
       JMP(LP)
```

(A) What are the values in R0, SP, BP and LP at the time execution was halted? Please express the values in hex or write "CAN'T TELL".

Value in R0: 0x 1 = ilog2(0)+1 **in SP: 0x** 24C

Value in BP: 0x 244 **in LP: 0x** 1A8

(B) Please fill in the values for the five blank locations in the stack trace shown on the right. Please express the values in hex.

Fill in values (in hex!) for 5 blank locations

(C) In the assembly language code for ilog2 there is the instruction "LD(BP,-12,R0)". If this instruction were rewritten as "LD(SP,NNN,R0)" what is correct value to use for NNN?

Correct value for NNN: -20

(D) In the assembly language code for ilog2, what is the address of the memory location labeled "xxx:"? Please express the value in hex.

Address of location labeled "xxx:": 0x 1B8

Values are in hex!

	5	x
	1A8	LP
	208	BP
	2	y
	5	R1
	2	x
	1A8	LP
	21C	BP
230	1	y
234	2	R1
238	1	x
23C	1A8	LP
240	230	BP
244 BP→	0	y
248	1	R1
24C SP→	0	