Compiler Optimization and Code Generation

Professor: Sc.D., Professor
Vazgen Melikyan

Course Overview

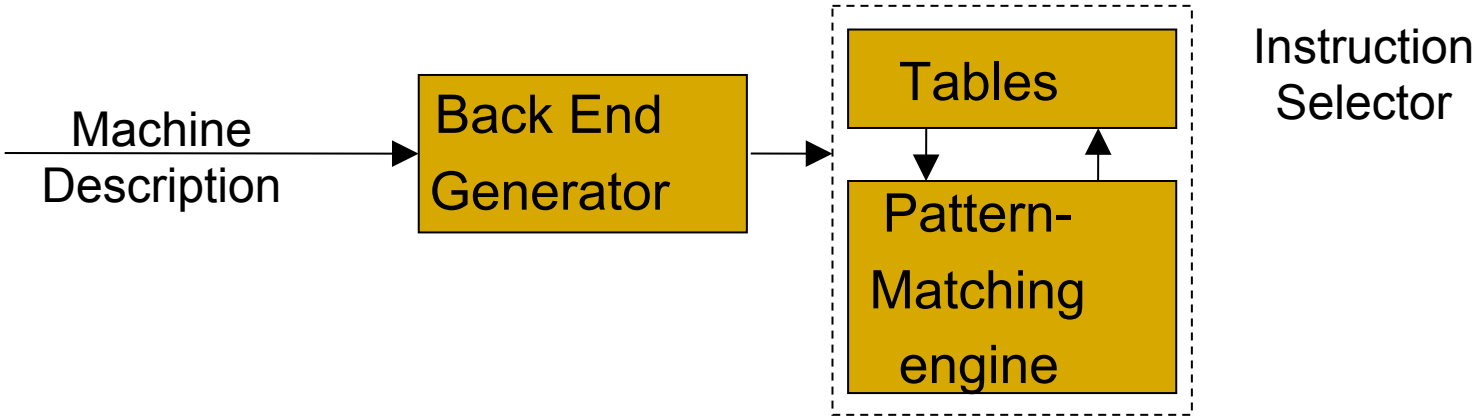
- Introduction: Overview of Optimizations
  - 1 lecture
- Intermediate-Code Generation
  - 2 lectures
- Machine-Independent Optimizations
  - 3 lectures
- Code Generation
  - 2 lectures

Code Generation

Machine Code Generation

- Input: intermediate code + symbol tables
  - In this case, three-address code
  - All variables have values that machines can directly manipulate
  - Assume program is free of errors
    - Type checking has taken place, type conversion done
- Output:
  - Absolute/relocatable machine code or assembly code
  - In this case, use assembly
  - Architecture variations: RISC, CISC, stack-based
- Issues:
  - Memory management, instruction selection and scheduling, register allocation and assignment

Retargetable Back End



- Build retargetable compilers
  - Isolate machine dependent info
  - Compilers on different machines share a common IR
    - Can have common front and mid ends
  - Table-based back ends share common algorithms
- Table-based instruction selector
  - Create a description of target machine, use back-end generator

Translating from Three-Address Code

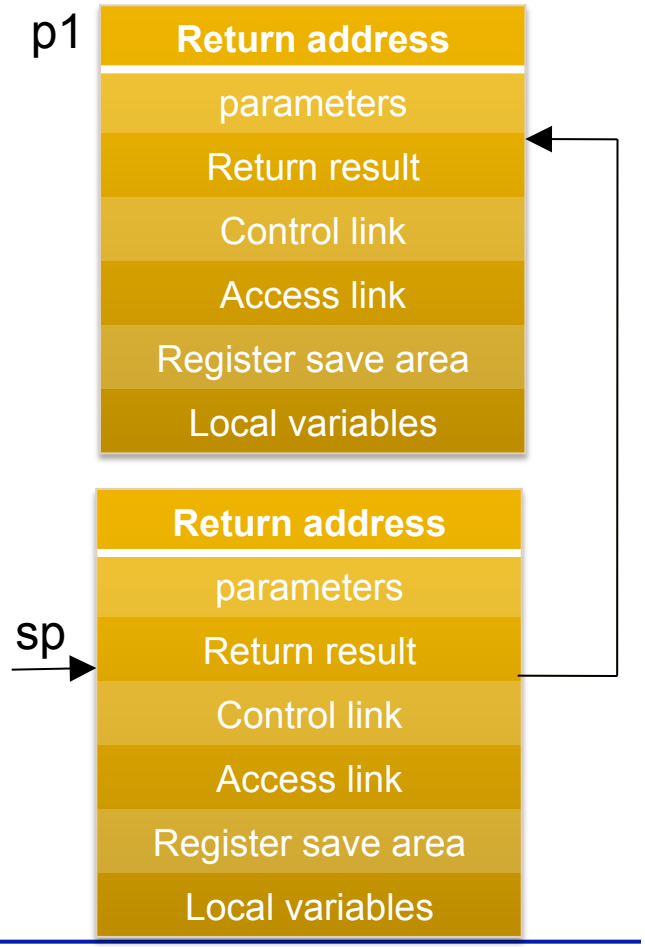
- No more support for structured control-flow
  - Function calls => explicit memory management and goto jumps
- Every three-address instruction is translated into one or more target machine instructions
  - The original evaluation order is maintained
- Memory management
  - Every variable must have a location to store its value
    - Register, stack, heap, static storage
  - Memory allocation convention
    - Scalar/atomic values and addresses => registers, stacks
    - Arrays => heap
    - Global variables => static storage

Assigning Storage Locations

- Compilers must choose storage locations for all values
  - Procedure-local storage
    - Local variables not preserved across procedural calls
  - Procedure-static storage
    - Local variables preserved across procedural calls
  - Global storage - global variables
  - Run-time heap - dynamically allocated storage
- Registers - temporary storage for applying operations to values
  - Unambiguous values can be assigned to registers with no backup storage

Function Call and Return

- At each function call
  - Allocate an new AR on stack
  - Save return address in new AR
  - Set parameter values and return results
  - Go to caller's code
    - Save SP and other regs; set AL if necessary
- At each function return
  - Restore SP and regs
  - Go to return address in caller's AR
  - Pop caller's AR off stack
- Different languages may implement this differently



Translating Function Calls

- Use a register SP to store address of activation record on top of stack
  - SP,AL and other registers saved/restored by caller
- Use C(Rs) address mode to access parameters and local variables

```
/* code for s */
Action1
Param 5
Call q, 1
Action2
Halt
……
/* code for q */
Action3
return
```

```
LD stackStart =>SP     /* initialize stack*/
……
108: ACTION1
128: Add SP,ssize=>SP  /*now call sequence*/
136: ST 160 =>*SP  /*push return addr*/
144: ST 5 => 2(SP) /* push param1*/
152: BR 300   /* call q */
160: SUB SP, ssize =>SP /*restore SP*/
168: ACTION2
190: HALT
     ……      /* code for q*/
300: save SP,AL and other regs
       ACTION3
restore SP,AL and other regs
400: BR *0(SP)  /* return to caller*/
```


Translating Variable Assignment

- Keep track of locations for variables in symbol table
  - The current value of a variable may reside in a register, a stack memory location, a static memory location, or a set of these
  - Use symbol table to store locations of variables
- Allocation of variables to registers
  - Assume infinite number of pseudo registers
  - Relocate pseudo registers afterwards

Statements	Generated code	Register descriptor	Address descriptor
t := a - b	LD a => r0 LD b => r1 SUB r0,r1=>r0	r0 contains t r1 contains b	t in r0 b in r1
u := t + c	LD c => r2 ADD r0,r2=>r0	r0 contains u r1 contains b r2 contains c	u in r0 b in r1 c in r2

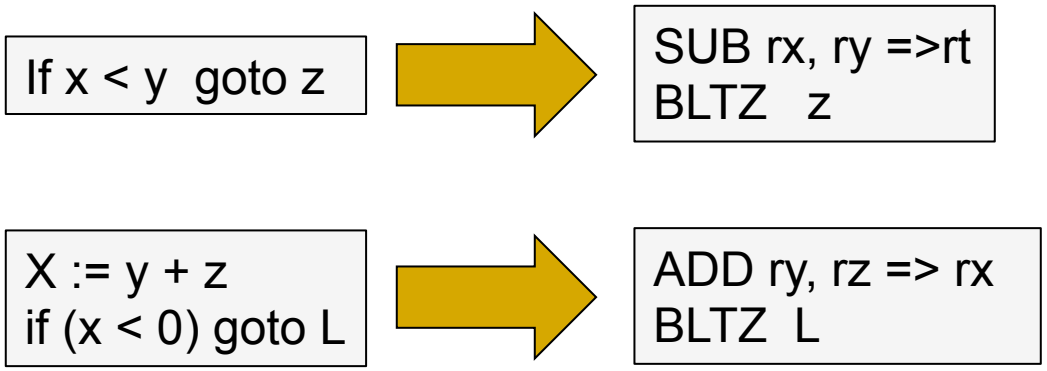
Translating Arrays

- Arrays are allocated in heap

Statement	i in register ‘ri’	i in memory ‘Mi’	i in stack
a := b[i]	Mult ri, elsize=>r1 LD b(r1)=>ra	LD Mi => ri Mult Ri,elsize=>r1 LD b(r1) =>ra	LD i(SP) => ri Mult ri,elsize=>r1 LD b(r1) =>ra
a[i] := b	Mult ri, elsize=>r1 ST rb => a(r1)	LD Mi => ri Mult Ri,elsize=>r1 ST rb => a(r1)	LD i(SP) => ri Mult ri,elsize=>r1 ST rb => a(r1)

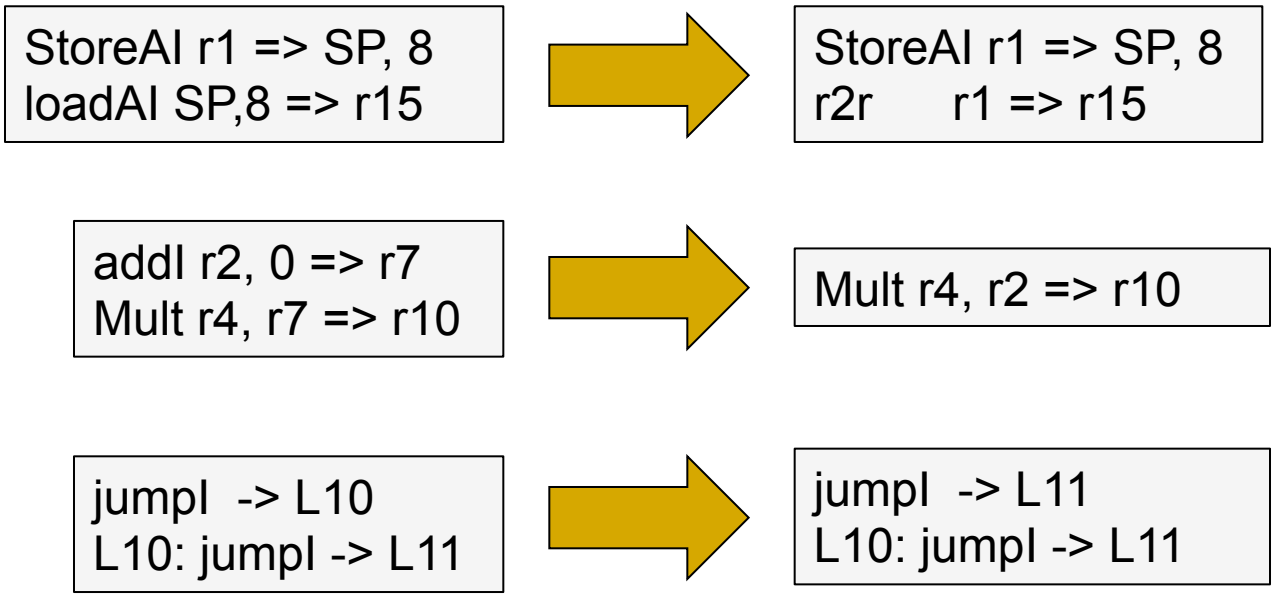
Translating Conditional Statements

- Condition determined after ADD or SUB



Peephole Optimization

- Use a simple scheme to match IR to machine code
  - Efficiently discover local improvements by examining short sequences of adjacent operations



Efficiency of Peephole Optimization

- Design issues
  - Dead values
    - May intervene with valid simplification
    - Need to be recognized expansion process
  - Control flow operations
    - Complicates simplifier: Clear window vs. special-case handling
  - Physical vs. logical windows
    - Adjacent operations may be irrelevant
    - Sliding window includes ops that define or use common values
- RISC vs. CISC architectures
  - RISC architectures makes instruction selection easier
- Additional issues
  - Automatic tools to generate large pattern libraries for different architectures
  - Front ends that generate LLIR make compilers more portable

Register Allocation

- Problem
  - Allocation of variables (pseudo-registers) to hardware registers in a procedure
- Features
  - The most important optimization
    - Directly reduces running time (memory access => register access)
  - Useful for other optimizations
    - E.g. CSE assumes old values are kept in registers
- Goals
  - Find an allocation for all pseudo-registers, if possible.
  - If there are not enough registers in the machine, choose registers to spill to memory

An Abstraction for Allocation and Assignment

- Two pseudo-registers interfere if at some point in the program they cannot both occupy the same register.
- Interference graph: an undirected graph, where
  - Nodes = pseudo-registers
  - There is an edge between two nodes if their corresponding pseudo-registers interfere
- What is not represented
  - Extent of the interference between uses of different variables
  - Where in the program is the interference

Register Allocation and Coloring

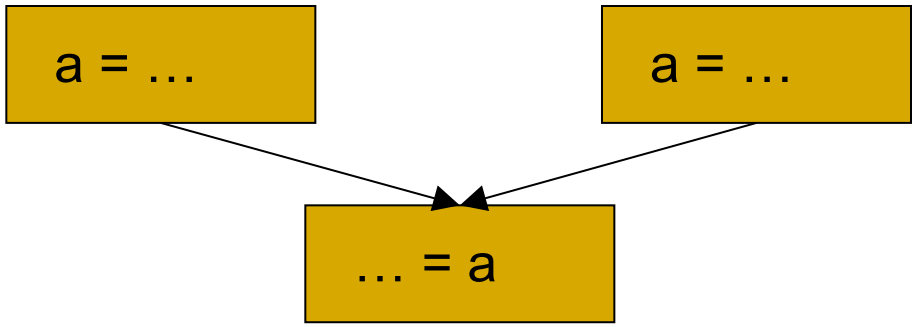
- A graph is n-colorable if:
  - Every node in the graph can be colored with one of the n colors such that two adjacent nodes do not have the same color.
- Assigning n register (without spilling) = Coloring with n colors
  - Assign a node to a register (color) such that no two adjacent nodes are assigned same registers(colors)
- Spilling is necessary = The graph is n-colorable

Algorithm

- Step 1: Build an interference graph
  - Refining notion of a node
  - Finding the edges
- Step 2. Coloring
  - Use heuristics to try to find an n-coloring
    - Success:
      - Colorable and we have an assignment
    - Failure:
      - Graph not colorable
      - Graph is colorable, but it is too expensive to color

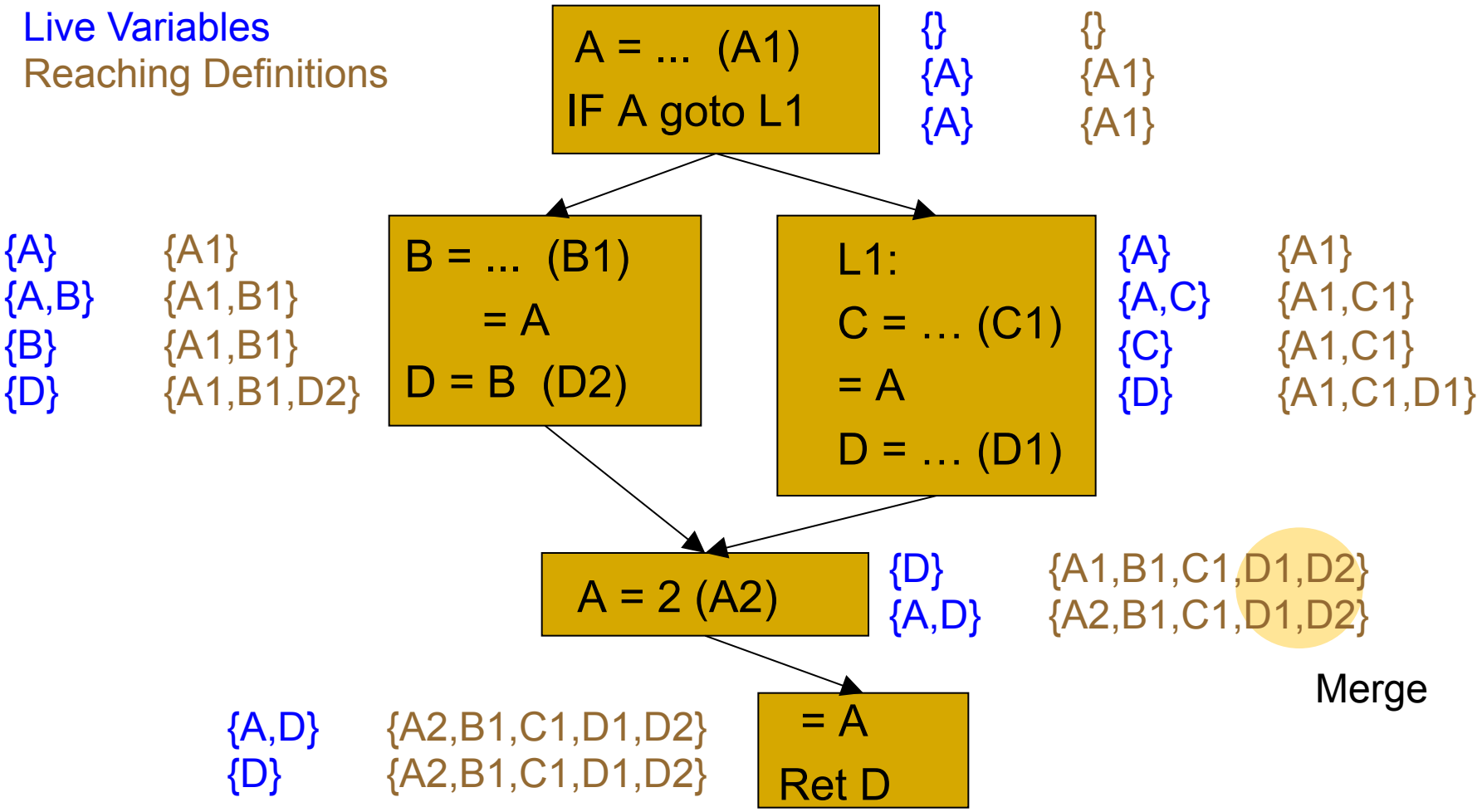
Live Ranges and Merged Live Ranges

- Motivation: to create an interference graph that is easier to color
  - Eliminate interference in a variable's "dead" zones
  - Increase flexibility in allocation
    - Can allocate same variable to different registers
- A live range consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.
- Two overlapping live ranges for the same variable must be merged



Example

Live Variables
Reaching Definitions



Merging Live Ranges

- Merging definitions into equivalence classes
  - Start by putting each definition in a different equivalence class
  - For each point in a program:
    - If (i) variable is live, and (ii) there are multiple reaching definitions for the variable, then:
      - Merge the equivalence classes of all such definitions into one equivalence class
- From now on, refer to merged live ranges simply as live ranges
  - Merged live ranges are also known as “webs”

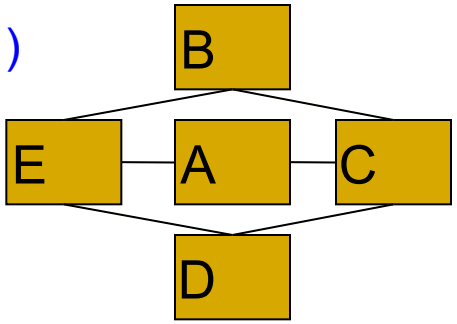
Edges of Interference Graph

- Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.
- Algorithm
  - At each point in the program enter an edge for every pair of live ranges at that point.
- An optimized definition & algorithm for edges:
  - Algorithm:
    - Check for interference only at the start of each live range
  - Faster
  - Better quality

Coloring

- Coloring for n > 2 is NP-complete
- Observations:
  - A node with degree < n can always color it successfully, given its neighbors' colors
- Coloring Algorithm
  - Iterate until stuck or done
    - Pick any node with degree < n
    - Remove the node and its edges from the graph
  - If done (no nodes left) reverse process and add colors

Example (n=3)



Note: degree of a node may drop in iteration

When Coloring Fails

- Using heuristics to improve its chance of success and to spill code

Build interference graph

Iterative until there are no nodes left
    If there exists a node v with less than n neighbor
        place v on stack to register allocate
    else
        v = node chosen by heuristics
          (least frequently executed, has many neighbors)
        place v on stack to register allocate (mark as spilled)
    remove v and its edges from graph

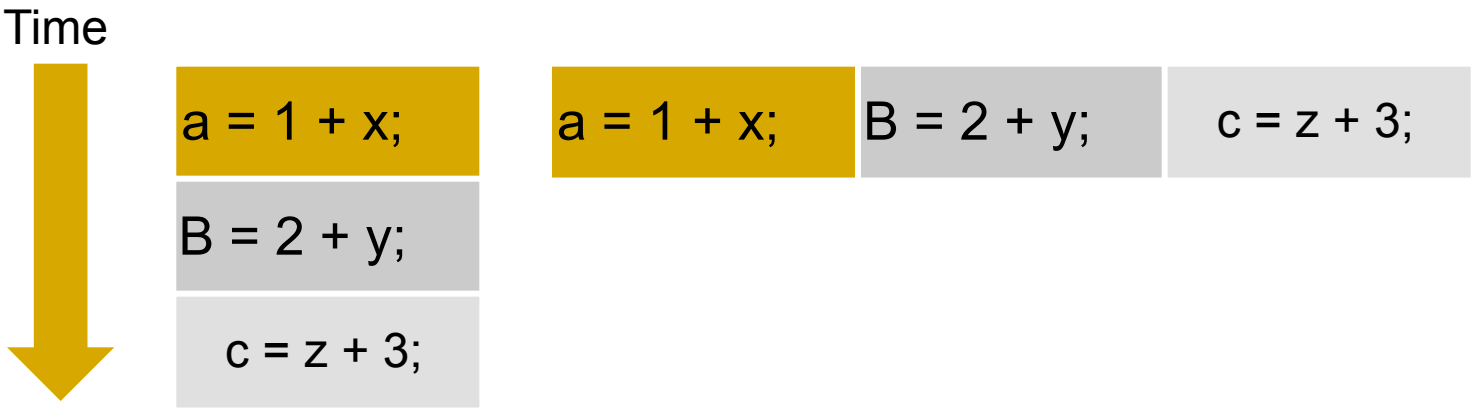
While stack is not empty
    Remove v from stack
    Reinsert v and its edges into the graph
    Assign v a color that differs from all its neighbors
    (guaranteed to be possible for nodes not marked as spilled)

Register Allocation: Summary

- Problem:
  - Find an assignment for all pseudo-registers, whenever possible.
- Solution:
  - Abstract on Abstraction: an interference graph
    - Nodes: live ranges
    - Edges: presence of live range at time of definition
  - Register Allocation and Assignment problems
    - Equivalent to n-colorability of interference graph
  - Heuristics to find an assignment for n colors
    - Successful: colorable, and finds assignment
    - Not successful: colorability unknown and no assignment

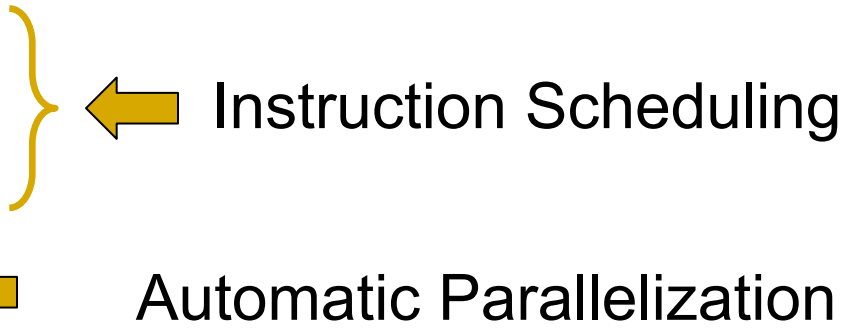
Instruction Scheduling: The Goal

- Assume that the remaining instructions are all essential: otherwise, earlier passes would have eliminated them
- The way to perform fixed amount of work in less time: execute the instructions in parallel



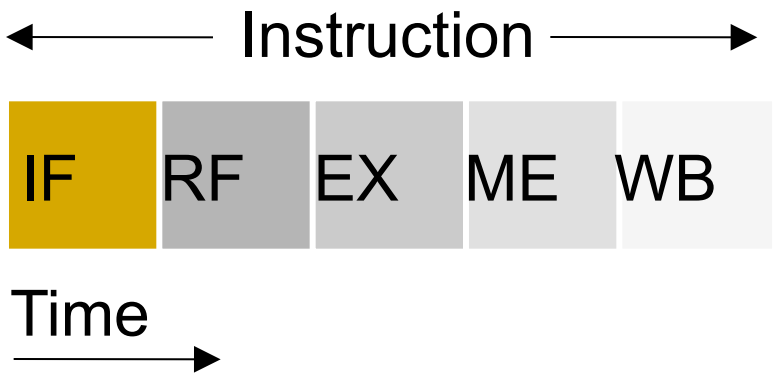
Hardware Support for Parallel Execution

- Three forms of parallelism are found in modern machines:
  - Pipelining
  - Superscalar Processing
  - Multiprocessing



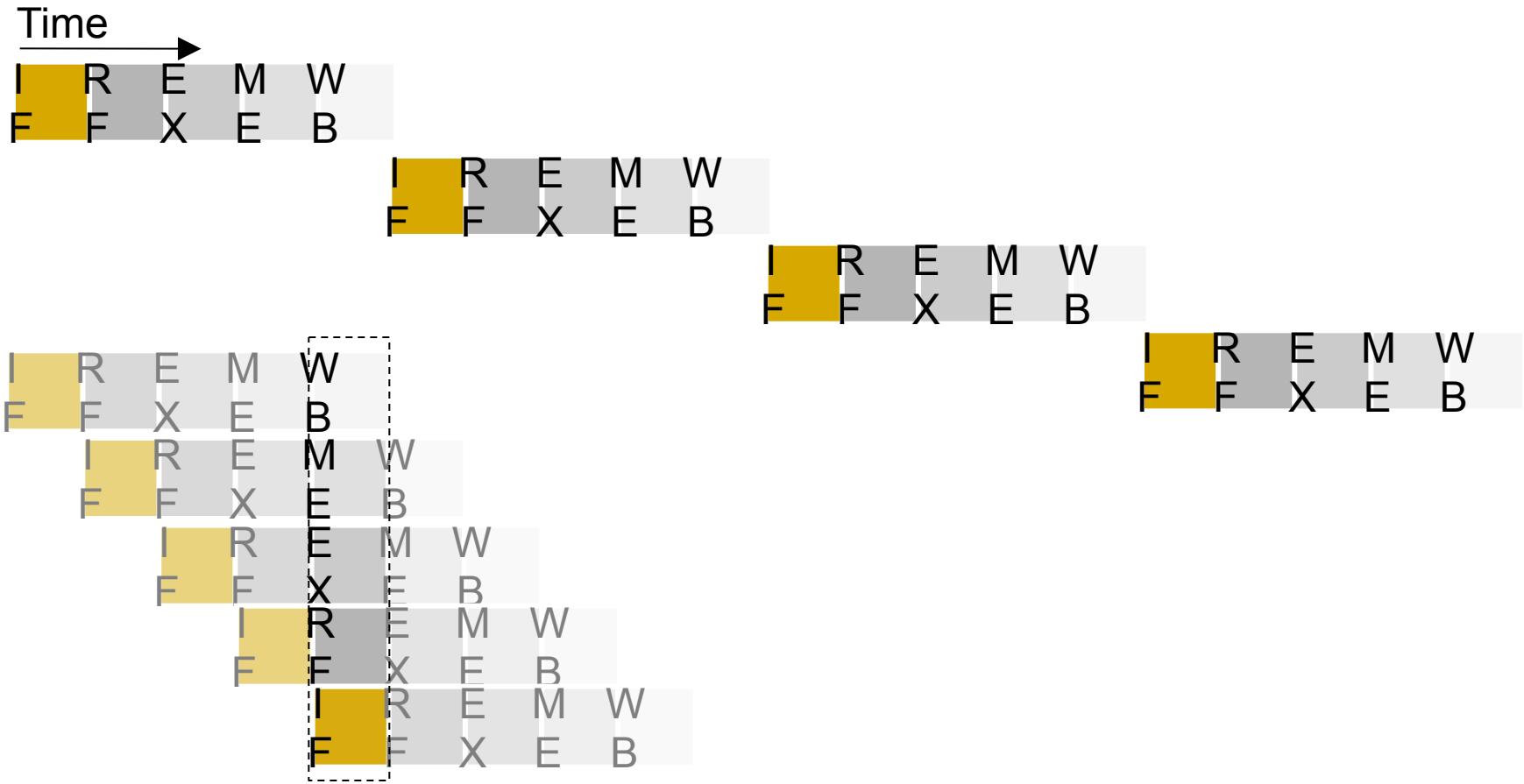
Pipelining

- Basic idea:
  - Break instruction into stages that can be overlapped
- Example:
  - Simple 5-stage pipeline from early RISC machines



IF = Instruction Fetch
RF = Decode & Register Fetch
EX = Execute on ALU
ME = Memory Access
WB = Write Back to Register File

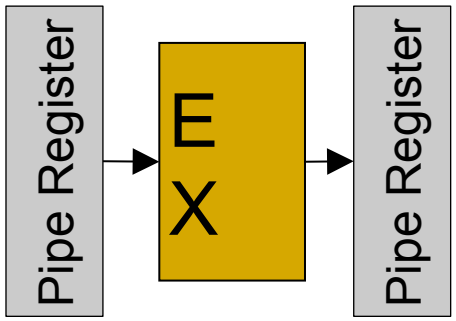
Pipelining Illustration



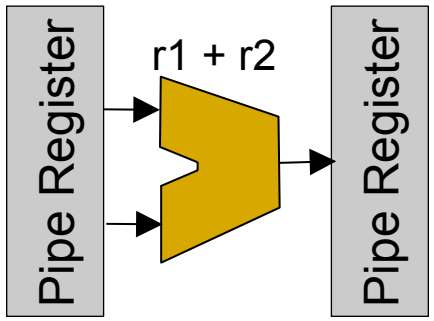
In a given cycle, each instruction is in a different stage

Beyond Pipelining: "Superscalar" Processing

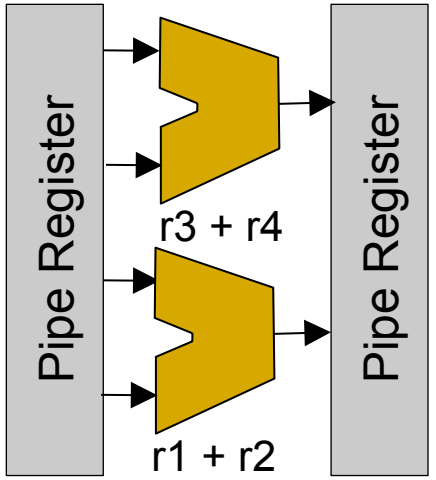
- Basic idea:
  - Multiple (independent) instructions can proceed simultaneously through the same pipeline stages
  - Requires additional hardware



Abstract Representation

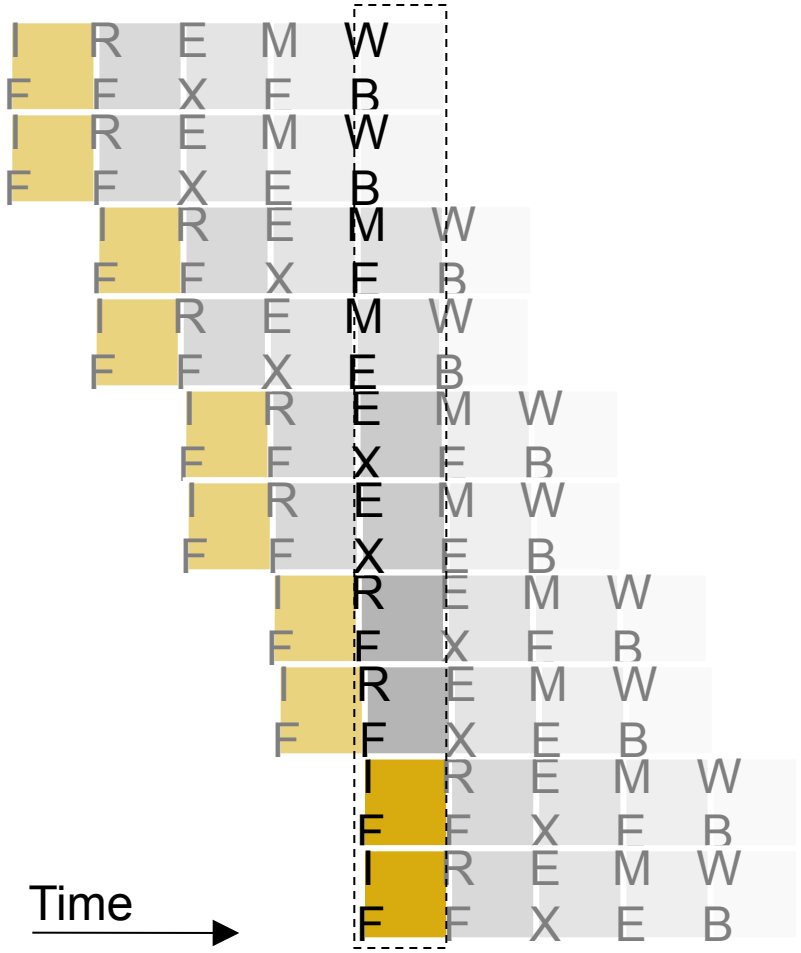


Hardware for Scalar pipeline 1 ALU



Hardware for 2 way superscalar 2 ALUs

Superscalar Pipeline Illustration



- Original (scalar) pipeline:
  - Only one instruction in a given pipe stage at a given time
- Superscalar pipeline:
  - Multiple instructions in the same pipe stage at the same time

Limitations upon Scheduling

- Hardware Resources
  - Processors have finite resources, and there are often constraints on how these resources can be used.
  - Examples:
    - Finite issue width
    - Limited functional units (FUs) per given instruction type
    - Limited pipelining within a given functional unit (FU)
- Data Dependences
  - While reading or writing a data location too early, the program may behave incorrectly.
- Control Dependences
  - Impractical to schedule for all possible paths
  - Choosing an expected path may be difficult

Predictable Success

Compiler Optimization and Code Generation
Lecture - 4
Developed By: Vazgen Melikyan