ECE 697J – Advanced Topics in Computer Networks

Microengine Programming I

11/18/03

Overview

- Lab 2: IP forwarding on IXP1200
  - Any problems with Part I?
- Microengine Assembler
  - Instructions
  - Preprocessor
  - Structured Programming Directives
- Lab 2: Classification on IXP1200
  - Simple packet classification

Microengine Assembler

- Assembly languages matches the underlying hardware
  - Intel developed “microengine assembly language”
- Assembly is difficult to program directly
  - Assembler supports higher-level statements
- High-level mechanisms:
  - Assembler directives
  - Symbolic register names and automated register allocation
  - Macro preprocessor
  - Pre-defined macros for common control structures
- Balance between low-level and higher-level programming

Assembly Language Syntax

- Instructions:

  `label:  operator  operands  token`

  – Operands and token are optional
  – Label: symbolic name as target for branch
  – Operator: single microengine instruction or high-level command
  – Operands and token: depend on operator
- Comments:
  – C-style: **`/* comment */`**
  – C++-style: **`// comment`**
  – ASM-style: **`; comment`**
  – Benefit of ASM style: remain with code after preprocessing
- Directives:
  – Start with "."

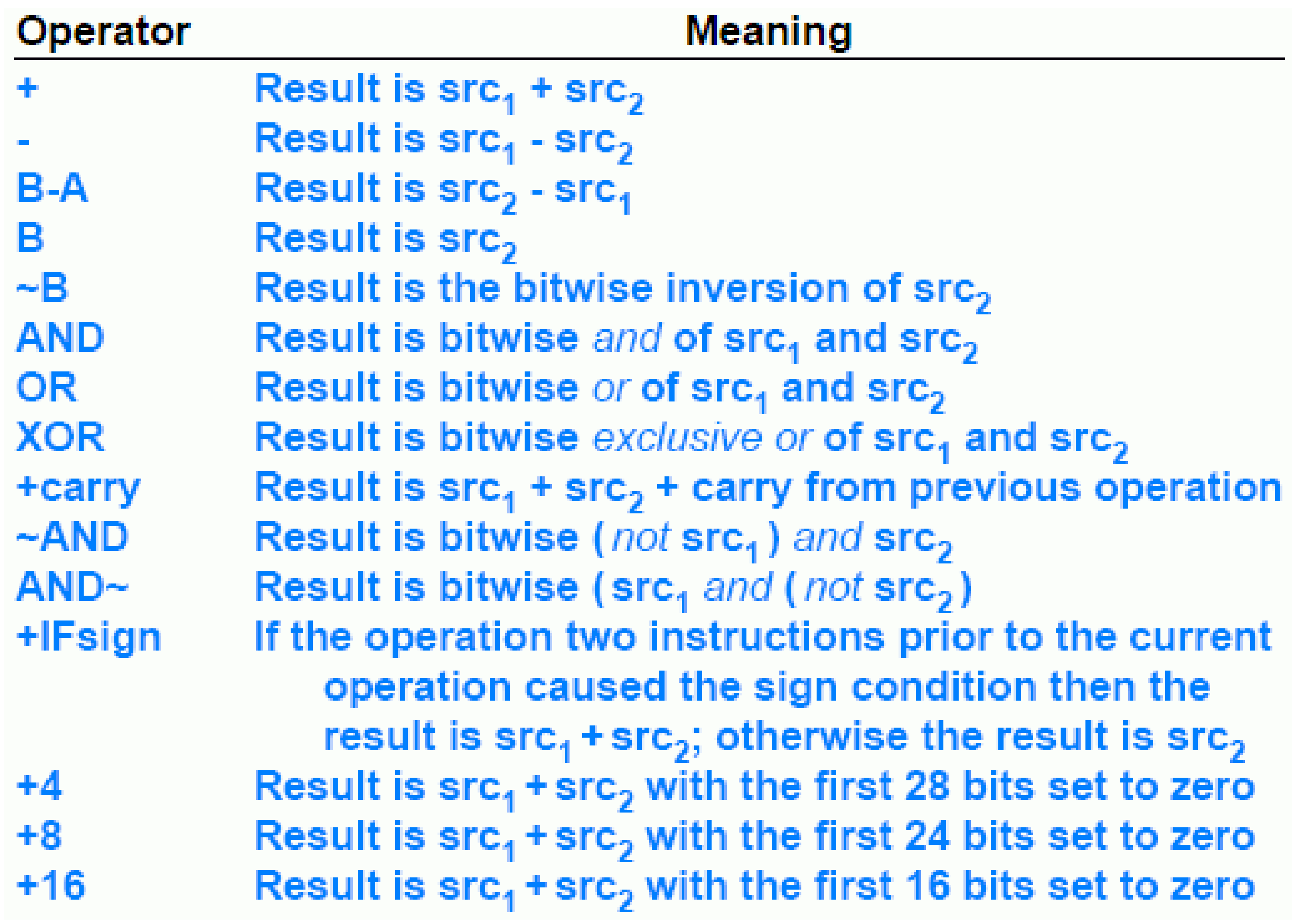
Operand Syntax

- Example: ALU instruction

  `alu [dst, src`$_1$`, op, src`$_2$`]`

  - **`dst`**: destination for result
  - **`src`**$_1$ and **`src`**$_2$: source values
  - **`op`**: operation to be performed
- Notes:
  - Destination register cannot be read-only (e.g., read transf. reg.)
  - If two source regs are used, they must come from different banks
  - Immediate values can be used
  - "**`--`**" indicates non-existing operand (e.g., source 2 for unary operation or destination)

ALU Operators



Operator	Meaning
+	Result is $src_1 + src_2$
-	Result is $src_1 - src_2$
B-A	Result is $src_2 - src_1$
B	Result is src_2
~B	Result is the bitwise inversion of src_2
AND	Result is bitwise *and* of src_1 and src_2
OR	Result is bitwise *or* of src_1 and src_2
XOR	Result is bitwise *exclusive or* of src_1 and src_2
+carry	Result is $src_1 + src_2$ + carry from previous operation
~AND	Result is bitwise (*not* src_1) *and* src_2
AND~	Result is bitwise (src_1 *and* (*not* src_2)
+IFsign	If the operation two instructions prior to the current operation caused the sign condition then the result is $src_1 + src_2$; otherwise the result is src_2
+4	Result is $src_1 + src_2$ with the first 28 bits set to zero
+8	Result is $src_1 + src_2$ with the first 24 bits set to zero
+16	Result is $src_1 + src_2$ with the first 16 bits set to zero

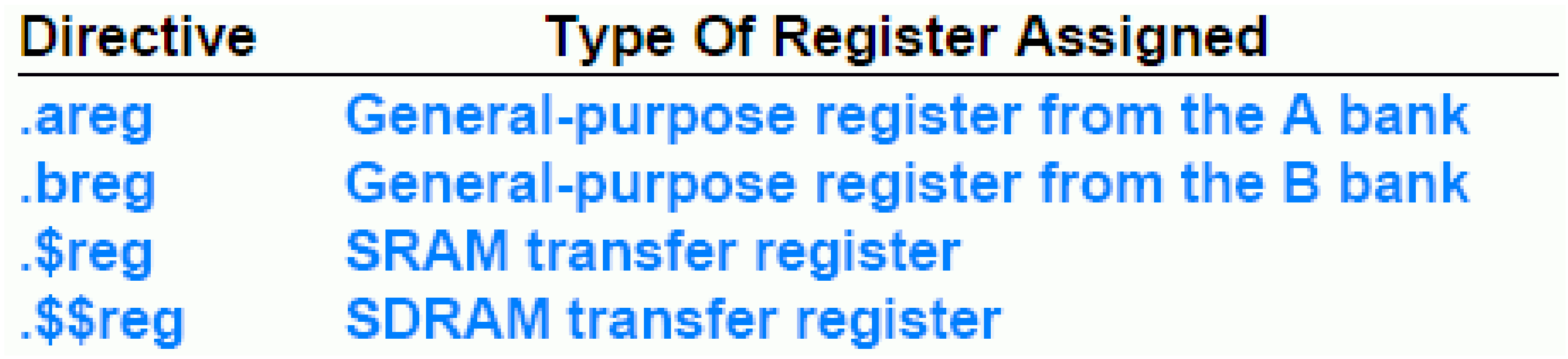
University *of* Massachusetts Amherst

Other Operators

- ALU shift/rotate:
  - **`alu_shf [dst, src`$_1$`, op, src`$_2$`, shift]`**
  - **`shift`** specifies right or left and shift or rotate (e.g., <<12, >>rot3)
- Memory accesses:
  - **`sram [direction, xfer_reg, addr`$_1$`, addr`$_2$`, count]`**
  - **`direction`** is “read” or “write”
  - **`addr`**$_1$ and **`addr`**$_2$ are used for base+offset and scaling
- Immediate:
  - **`immed [dst, ival, rot]`**
  - Immediate has upper 16 bit all 0 or all 1
  - Rotation is “0”, “<<8”, or “<<16”
  - Also direct access to individual bytes/words: immed_b2, immed_w1

Symbolic Register Names

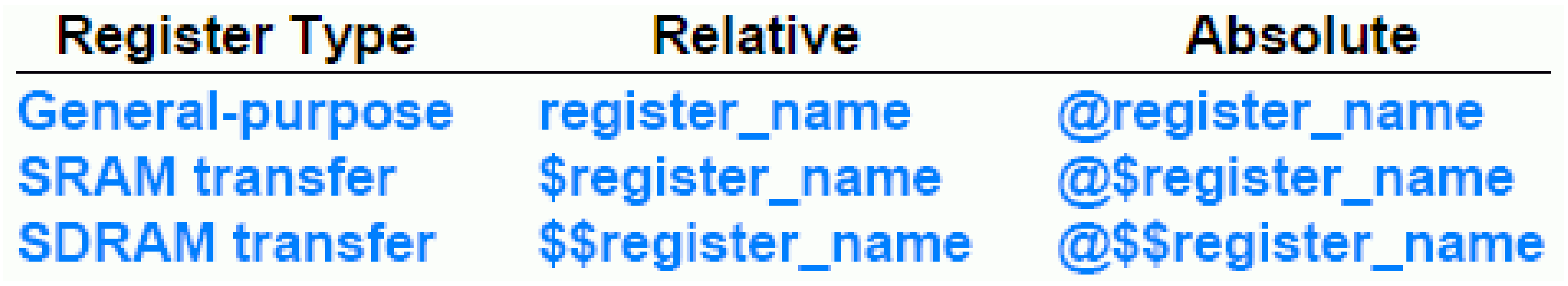
- Assembler supports automatic register allocation
  - Either entirely manual or automatic – no mixture possible
- Symbolic register names:
  - **`.areg   loopindex   5`**
  - Assigns the symbolic name “loopindex” to register 5 in bank A
- Other directives:



Directive	Type Of Register Assigned
.areg	General-purpose register from the A bank
.breg	General-purpose register from the B bank
.$reg	SRAM transfer register
.$$reg	SDRAM transfer register

Register Types and Syntax

- Register names with relative and absolute addressing:

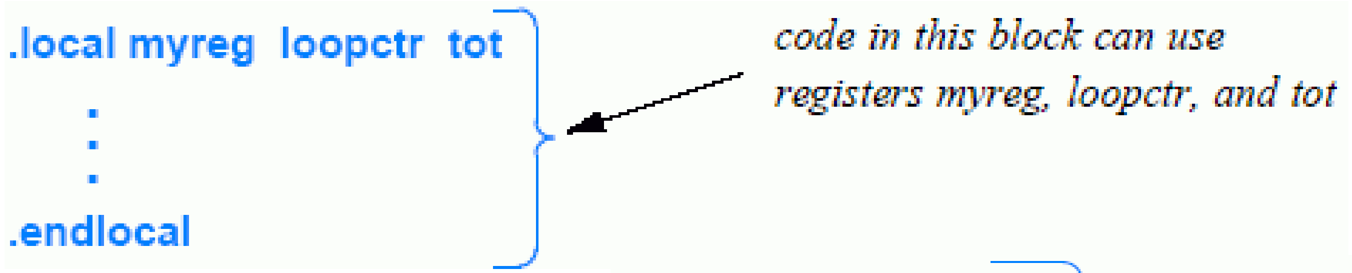


Register Type	Relative	Absolute
General-purpose	register_name	@register_name
SRAM transfer	$register_name	@$register_name
SDRAM transfer	$$register_name	@$$register_name

- Note: read and write transfer registers are separate
  – You cannot read a value after you have written it to a xfer reg
- Also: some instruction sequences impossible:
  – Z <- Q + R
  – Y <- R + S
  – X <- Q + S

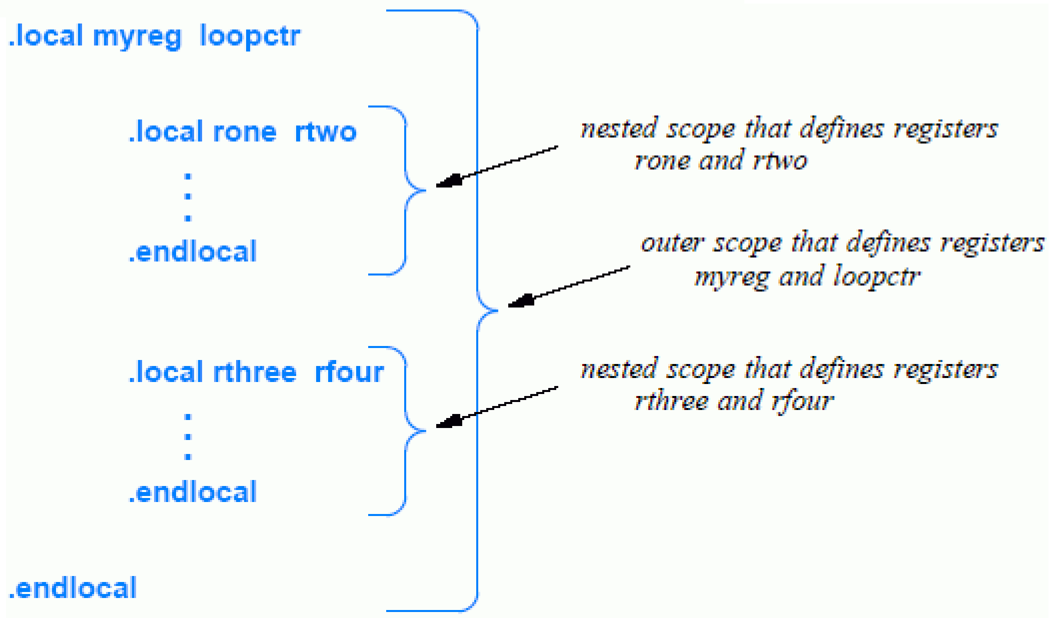
Scoping

- Scopes define regions where variable names are valid
  - .local directive:



```
.local myreg  loopctr  tot
      :
      :
.endlocal
```

- Outside scope registers can be reused
- Scopes can be nested
  - Names are "shadowed"



```
.local myreg  loopctr

    .local rone  rtwo
      :
    .endlocal

    .local rthree  rfour
      :
    .endlocal

.endlocal
```

nested scope that defines registers
rone and rtwo
outer scope that defines registers
myreg and loopctr
nested scope that defines registers
rthree and rfour

Macro Preprocessor

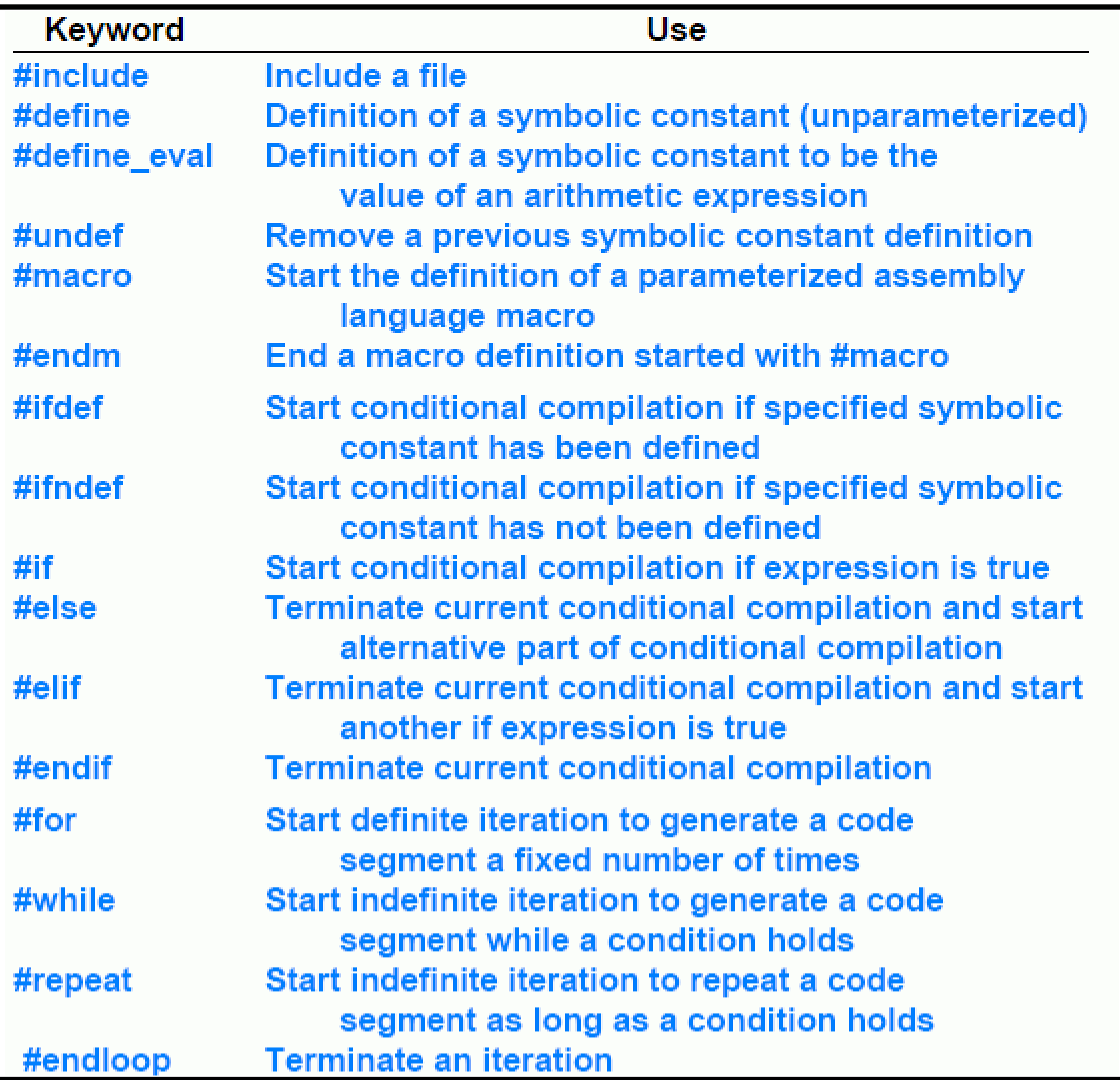
- Preprocessor functionality:
  - File inclusion
  - Symbolic constant substitution
  - Conditional assembly
  - Parameterized macro expansion
  - Arithmetic expression evaluation
  - Iterative generation of code
- Macro definition

  -
    ```
    #macro name [parameter1, parameter2, …]
           lines of text
    #endm
    ```

Macro Example

- Example for a=b+c+5:
  - `#macro add5 [a, b, c]`
    ```
    #macro add5 [a, b, c]
           .local tmp
                 alu[tmp, c, +, 5]
                 alu[a, b, +, tmp]
           .endlocal
    #endm
    ```
- Problems when tmp variable is overloaded:
  - **`add5[x, tmp, y]`**
  - Why?
- One has to be careful with marcos!

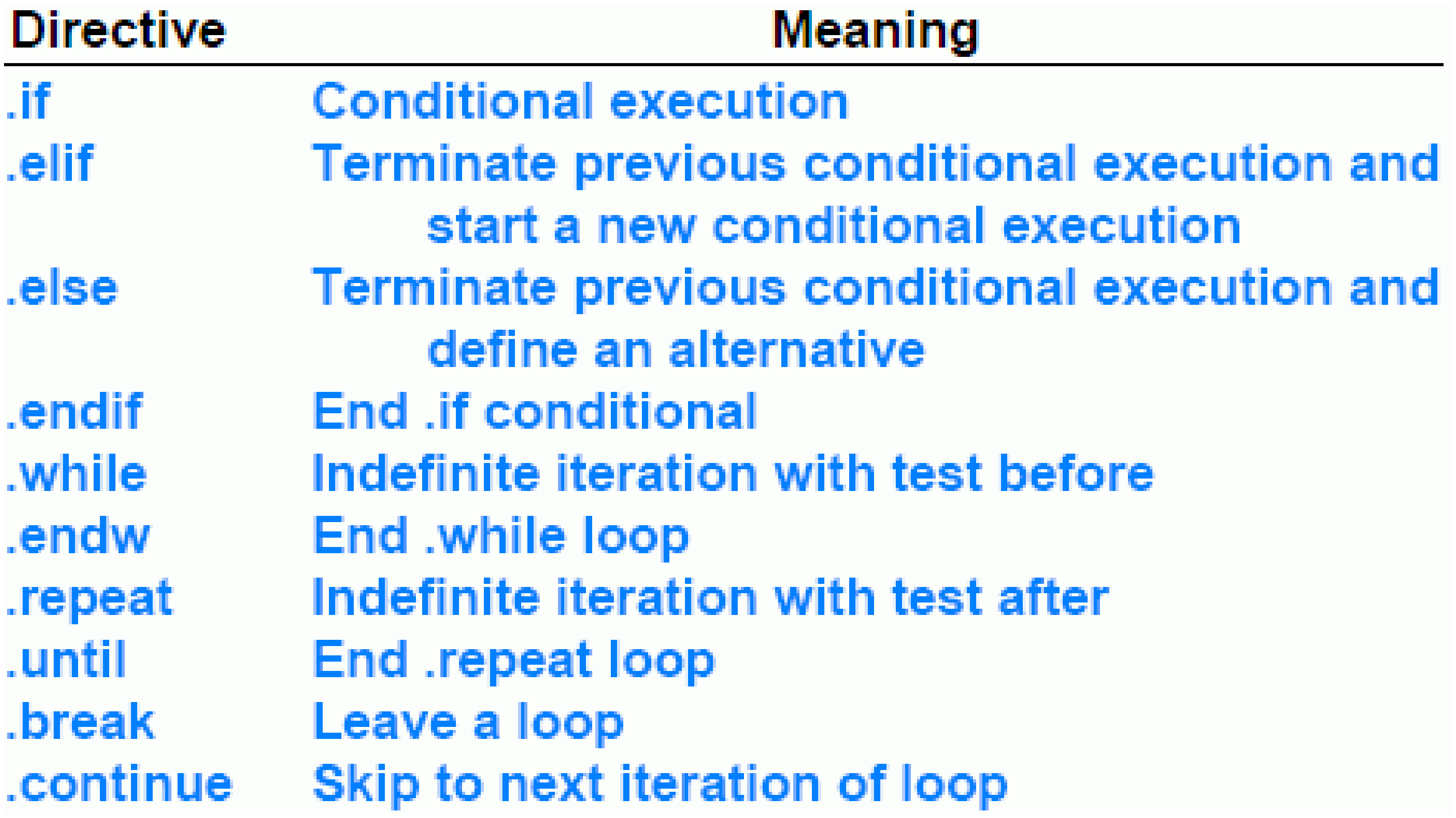
Preprocessor Statements



Keyword	Use
#include	Include a file
#define	Definition of a symbolic constant (unparameterized)
#define_eval	Definition of a symbolic constant to be the value of an arithmetic expression
#undef	Remove a previous symbolic constant definition
#macro	Start the definition of a parameterized assembly language macro
#endm	End a macro definition started with #macro
#ifdef	Start conditional compilation if specified symbolic constant has been defined
#ifndef	Start conditional compilation if specified symbolic constant has not been defined
#if	Start conditional compilation if expression is true
#else	Terminate current conditional compilation and start alternative part of conditional compilation
#elif	Terminate current conditional compilation and start another if expression is true
#endif	Terminate current conditional compilation
#for	Start definite iteration to generate a code segment a fixed number of times
#while	Start indefinite iteration to generate a code segment while a condition holds
#repeat	Start indefinite iteration to repeat a code segment as long as a condition holds
#endloop	Terminate an iteration

Structured Programming Directives

- Structured directives are similar to control statements:



Directive	Meaning
.if	Conditional execution
.elif	Terminate previous conditional execution and start a new conditional execution
.else	Terminate previous conditional execution and define an alternative
.endif	End .if conditional
.while	Indefinite iteration with test before
.endw	End .while loop
.repeat	Indefinite iteration with test after
.until	End .repeat loop
.break	Leave a loop
.continue	Skip to next iteration of loop

Example

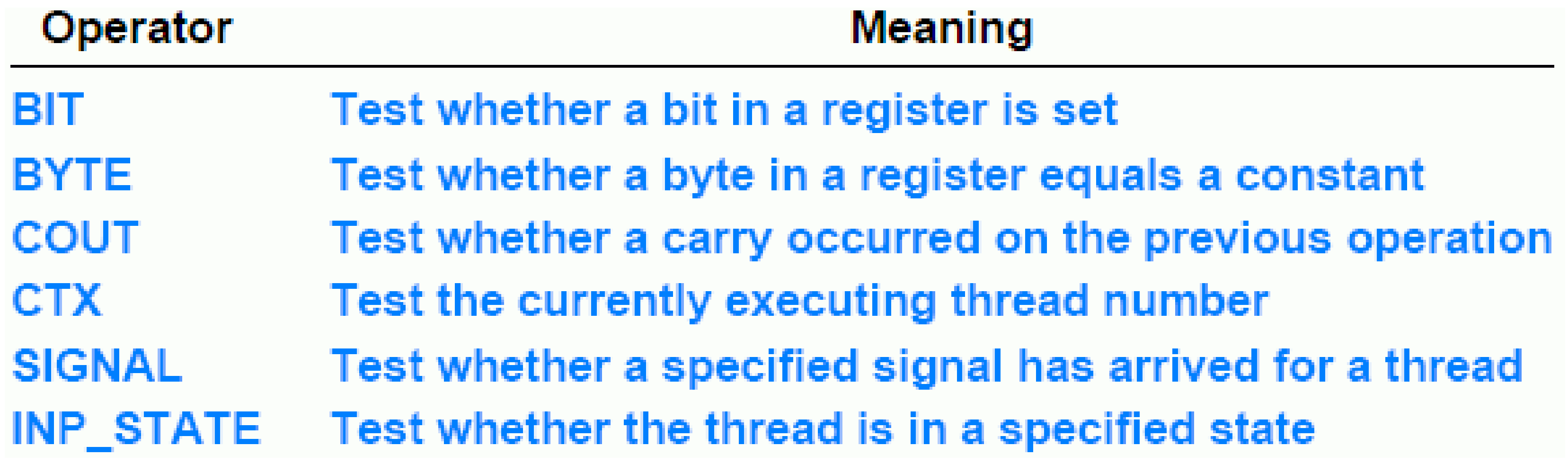
- If statement with structured directives:

  -
    ```
    .if ( conditional_expression )
           /* block of microcode */
    .elif ( conditional_expression )
           /* block of microcode */
    .else
           /* block of microcode */
    .endif
    ```
- While statement:

  -
    ```
    .while ( conditional_expression )
           /* block of microcode */
    .endw
    ```
- Very useful and less error-prone than hand-coding

Conditional Expressions

- Conditional expressions may have C-language operators
  - Integer comparison: <, >, <=, >=, ==, !=
  - Shift operator: <<, >>
  - Logic operators: &&, ||
  - Parenthesis: (,)
- Additional test operators



Operator	Meaning
BIT	Test whether a bit in a register is set
BYTE	Test whether a byte in a register equals a constant
COUT	Test whether a carry occurred on the previous operation
CTX	Test the currently executing thread number
SIGNAL	Test whether a specified signal has arrived for a thread
INP_STATE	Test whether the thread is in a specified state

Context Switches

- Instructions that cause context switches:
  - `ctx_arb` instruction
  - Reference instruction
- `ctx_arb` instruction:
  - One argument that specifies how to handle context switch
  - `voluntary`
  - `signal_event` – waits for signal
  - `kill` – terminates thread permanently
- Reference instruction to memory, hash, etc.
  - One argument
  - `ctx_swap` – thread surrenders control until operation completed
  - `sig_done` – thread continues and is signaled completion

Indirect References

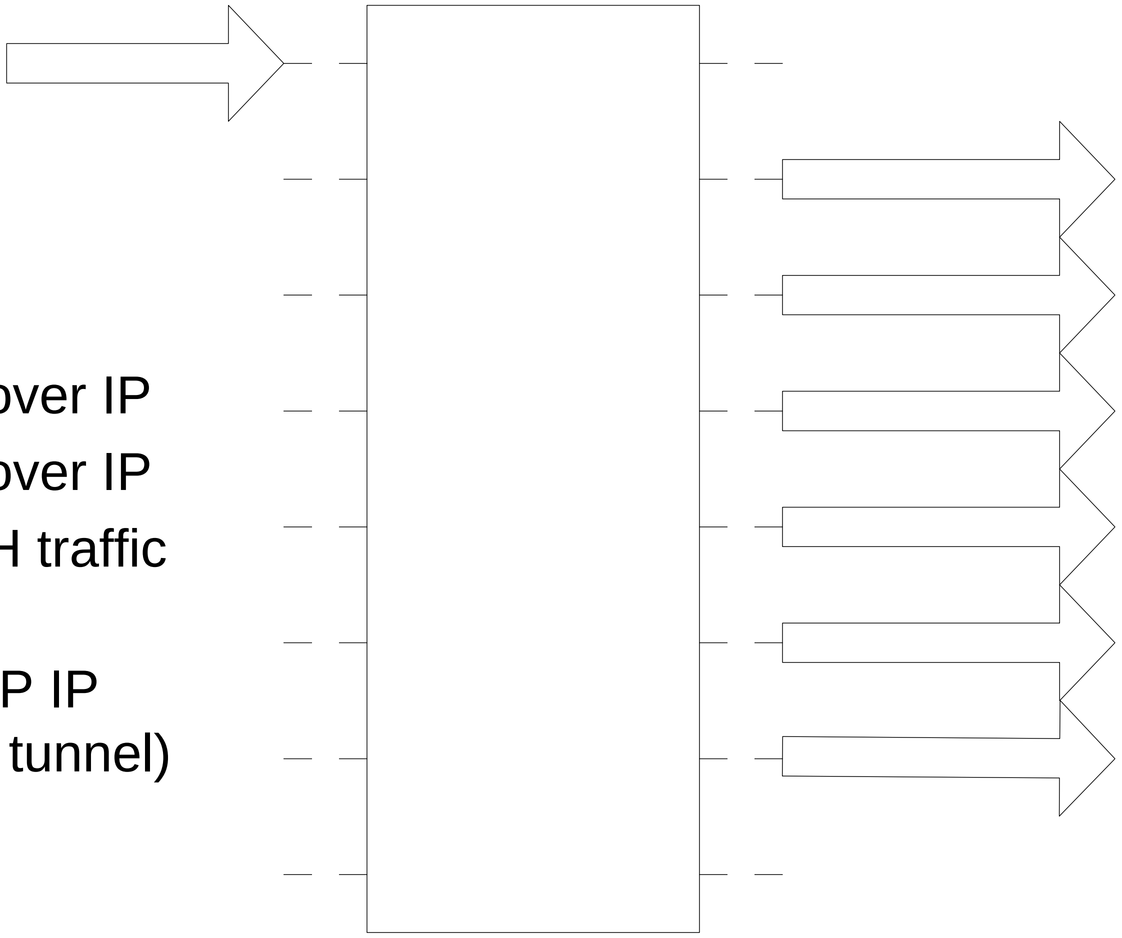
- Sometimes memory addresses are not known at compile time
  - Indirect references use result of ALU instruction to modify immediately following reference
  - *“Unlike the conventional use of the term [indirect reference], Intel’s indirect reference mechanism does not follow pointers; the terminology is confusing at best.”* ☺
- Indirect reference can modify:
  - Microengine associated with memory reference
  - First transfer register in a block that will receive result
  - The count of words of memory to transfer
  - The thread ID of the hardware thread executing the instruction
- Bit patterns specifying operation and parameter must be loaded into ALU
  - Uses operation without destination: **`alu_shf[--,--,b,0x13,<<16]`**
  - Reference: **`scratch[read,$reg0,addr1,addr2,0],indirect_ref`**

Transfer Registers

- Memory transfers need contiguous registers
  - Specified with **`.xfer_order`**
  - **`.local $reg1 $ref2 $ref3 $ref4`**
    `.xfer_order $reg1 $reg2 $reg3 $reg4`
- Library macros for transfer register allocation
  - Allocations: **`xbuf_alloc[]`**
  - Deallocation: **`xbuf_free[]`**
  - Example: **`xbuf_alloc[$$buf,4]`** allocates **`$$buf0, …, $$buf3`**
- Allocation is based on 32-bit chunks
  - Transfer of 2 SDRAM units requires 4 transfer registers

Lab 2 – Part II

- Packet Classification
- Traffic types:
  - ARP traffic
  - UDP over IP traffic
  - Web traffic over TCP over IP
  - SSH traffic over TCP over IP
  - Non-web and non-SSH traffic over TCP over IP
  - Non-TCP and non-UDP IP traffic (e.g., IP-over-IP tunnel)



Classification Code

```
// START ECE 697J CLASSIFICATION
xbuf_extract(ip_upp_pro, $pkt_buf_ip, BYTEOFFSET0,
                        IP_UPPER_LAYER_PROTOCOL);
.if (ip_upp_pro == TCP_PACKET)
  xbuf_extract(tcp_dport, $pkt_buf_ip, BYTEOFFSET18,
                                    TCP_DEST_PORT);
  .if (tcp_dport == TCP_SSH)
      move(output_intf, 0x0000008);
  .else
      move(output_intf, 0x00000008);
  .endif
.else
  move(output_intf, 0x00000000);
.endif
// END ECE 697J CLASSIFICATION
```

Lab 2 – Part II Questions

- Extend the given forwarding code to implement the classification as described above.
- Determine the traffic mix. What fraction of the traffic belongs to each of the six classes?
- Use the execution coverage window in the simulator to verify that the instruction coverage of your classifier matches the traffic mix results.
- Assume that the classification step was really critical for performance. In your implementation, you have a choice of making classification decisions in different orders (e.g., check for UDP packets before checking the type of TCP packet etc.). In what order should packets be classified given the traffic mix in this example? In general, if the traffic mix is known, in what order should classification be done?

Final Projects

- Ideas for final projects:
  - Implement a packet filter on IXP1200 hardware
    - E.g., don't forward telnet packets, but ssh packets
  - Analysis of memory contention on IXP1200
    - Write code to generate different amounts of load on memory
    - Analyze memory latency distribution and model it
  - Packet forwarding processing analysis
    - Count number of instructions spent on various steps of forwarding
    - Analyze impact of different # of uEs and threads
    - Compare to layer 2 bridging
  - Anything else?
- Project report ~15 pages with many interesting graphs and illustrations
- Final presentation: 20-30 minutes on 12/9/03

Next Class

- Microengine Programming II
  - Read chapter 25
  - Ramu will give lecture
- Help Session for Lab 2 – Part II
  - Ning will answer any questions (except “What is the solution?”)
- Lab 2 – Part II due 11/25
- Questions
  - What do you want to do for lab 3?
  - Do you want to change grade percentage from 20% labs and 40% final project to 30%/30%?