An ADL for Functional Specification of IA32

Wei Qin, Boston University

Intel Collaborators
Asa Ben-Tzur, Boris Gutkovich

June 30, 2008
Haifa, Israel

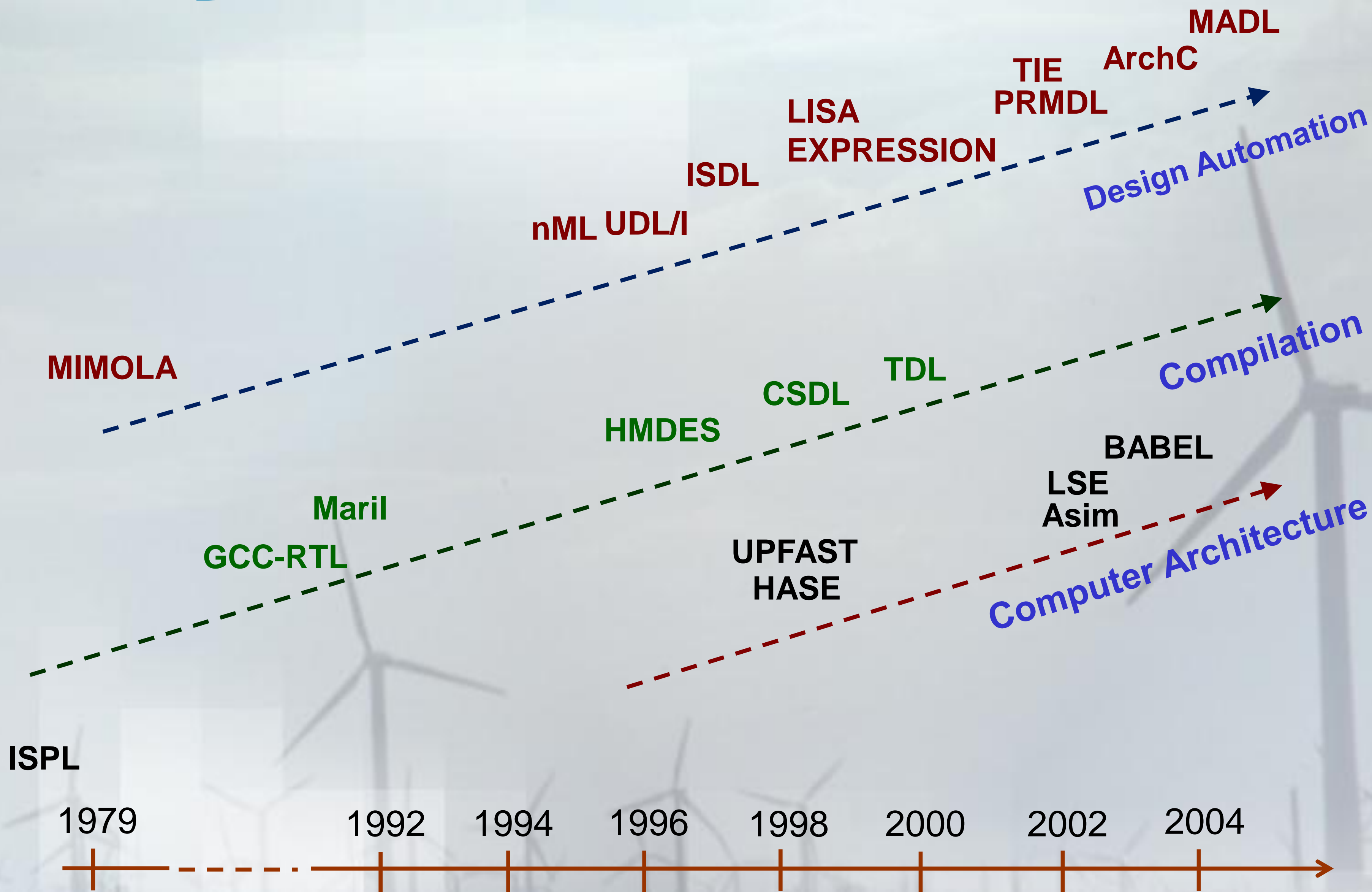
Motivation

- Development tools need ISA information
  - Commonly encoded in mixed formats
    - Natural language, C/C++, XML, etc.
  - Problems
    - Redundancy, consistency, reusability
- An ADL custom-made for IA32
  - Concise and elegant — minimal redundancy, simplicity
  - Usable — engineers understand without special training
  - Flexible — can describe all important features of IA32
  - Declarative — easy to reason about by the compiler
  - Extensible — support multiple applications

Outline

- ADL background
  - Existing ADLs
  - Challenges to model IA32
- Features of proposed ADL
- Semantic model
- Applications
- Conclusion

History of ADL



Yet Another ADL?

- Important challenges remain
  - Most ADLs focus on regular architectures
    - RISC, DSP
  - Most ADLs focus on instructions, but ignore
    - Modes of operation
    - Virtual memory mapping
    - Exception
  - No existing ADL fully covers IA32

Challenges to Model IA32

- Various modes of operation
- Asymmetric usage of registers
- Sophisticated addressing modes
- 16-bit and 32-bit operand/address sizes
- Complex instruction format with prefixes
- Complex instruction behaviors including SIMD
- Three memory models, paging, segmentation
- Side-effects and exceptions

Elements of the ADL

- General declarations
  - Include, alias
  - Data types
  - Function
- Physical storage declaration
  - Registers
  - Memory Views
- Operand declaration
- Instruction declaration
- Environment: context guarding declarations

Type System

- Boolean
  - E.g. exception condition
- Enumeration
  - Useful for readability
- Fixed-width integer
  - Supports bit manipulation
  - Supports the definition of fields
  - Supports inheritance
- Tuple
  - E.g. a logical address: (selector, offset)
  - Useful for passing values between operands/instructions
- Array
  - For register/memory

Fixed Width Integer Example

- Example: segment descriptor types

```
typedef uint<64> {
    base      : uint<24> = {63:56} {39:16},
    seg_limit : uint<20> = {51:48} {15:0},
    ……
    p         : uint<1>  = {47},
    dpl       : uint<2>  = {46:45},
    s         : uint<1>  = {44},
    type      : uint<4>  = {43:40},
} seg_des_t;

typedef seg_des_t {
    s = 0,                 // constrain the value of s
    t = 0     : {43},      // bit type can be omitted
    e         : {42},
    w         : {41},
    a         : {40},
} data_seg_des_t;
```

Register

- Register as scalar or array

```
register eflags  :eflags_t;/* EFLAGS */
register iregs[8]:ireg_t;  /* EAX,ECX,EDX,EBX,etc.*/
```

- Overlapping via fields

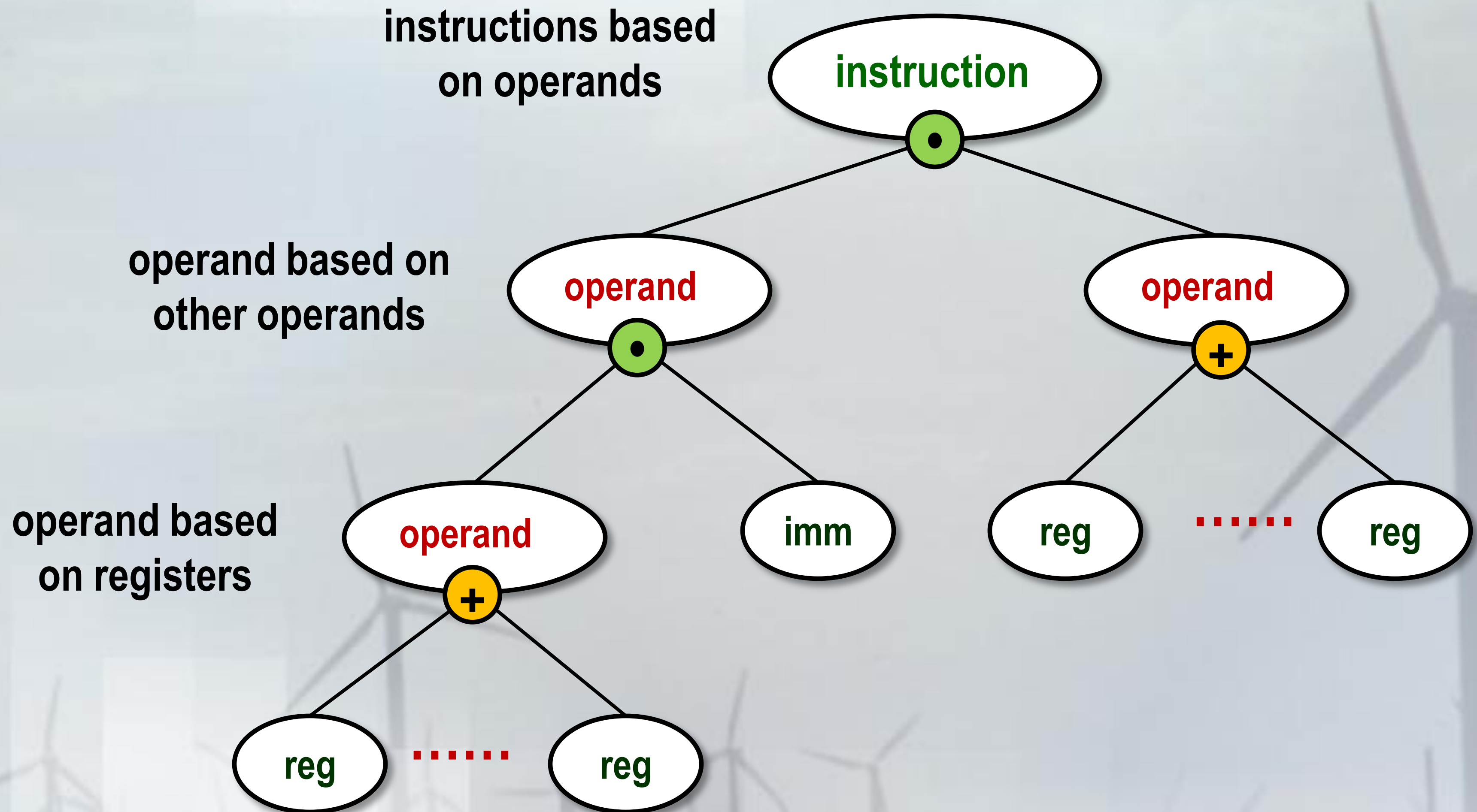
```
typedef uint<32> {
    half      : uint<16> = {15:0},
    hquart    : uint<8>  = {15:8},
    lquart    : uint<8>  = {7:0},
} ireg_t;
```

  - AL as `iregs[0].lquart`
  - AX as `iregs[0].half`
  - EAX as `iregs[0]`

- Alias for abbreviation

```
alias AL iregs[0].lquart;
```

Hierarchy of Description



Operand

- Each operand defines three attributes
  - Binary encoding, assembly syntax, semantics
- Operand Example

```
operand r8 : uint<8> {          /* 8 bit registers */
  option syntax = "AL", coding = 000, semantics = AL;
  option syntax = "CL", coding = 001, semantics = CL;
  ......
}
```

- Can build on child operands

```
operand rimm8(r8, imm8) : uint<8> {
  option syntax = "$1 + $2", coding = 00 $1 $2,
         semantics = $1+$2;
  option syntax = "$1 - $2", coding = 01 $1 $2,
         semantics = $1-$2;
}
```

Instruction

- Instruction also defines syntax, coding, semantics

```
instruction add_rm8 (RM8, r8) {
    syntax = "ADD $1, $2",
    coding = 00000000 $1.$1 $2 $1.$2,
    semantics = { $1 = fadd8($1, $2);}
}
```

- Expand AND/OR graph result in many instances
- Environment considered when expanding

Environment

- ADL is instruction centric
  - Same for all other ADLs
  - How to deal with issues that are non-local?
    - Architecture generation, processor Mode
    - Code segment size attributes
- Use environment to help
  - Operand/instruction guarded by environments
  - Change of environment may change meaning
  - Prefixes treated as environment

Environment Examples

- **Size attribute of code segment**

```
#if (seg_size==_32_bit) || (seg_size==_16_bit && defined(prefix_g4))
// define 32-bit version "RM" operand here
……
#else
// define 16-bit version "RM" operand here
……
#endif
```

- **Segment selector prefix**

```
operand RM_mem (RM_00 || RM_01 || RM_10 ) {
#if defined(prefix_g2) && prefix_g2==_2EH
  option syntax = "CS:$1", coding = $1, semantics = logical_mem[(CS, $1.$2)];
#elif defined(prefix_g2) && prefix_g2==_36H
  ……
#else
  option syntax = $1, coding = $1, semantics = logical_mem[($1.$1, $1.$2)];
#endif
}
```

Memory Modeling

- **Memory declaration**
  - Address type
  - Optional target, mapping function

```
memory physical_mem : physical_addr_t;
memory linear_mem : linear_addr_t = {
    target  = physical_mem,
    mapping = paging;
}
memory logical_mem : logical_addr_t = {
    target  = linear_mem,
    mapping = logical_to_linear;
}
```

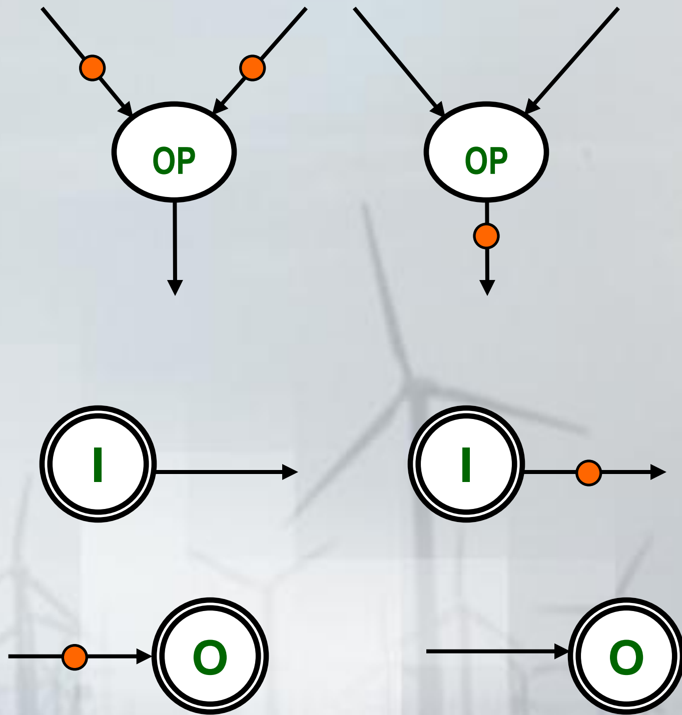
- Accessing memory using [ ]
  - Size and access type (read/write) depends on context

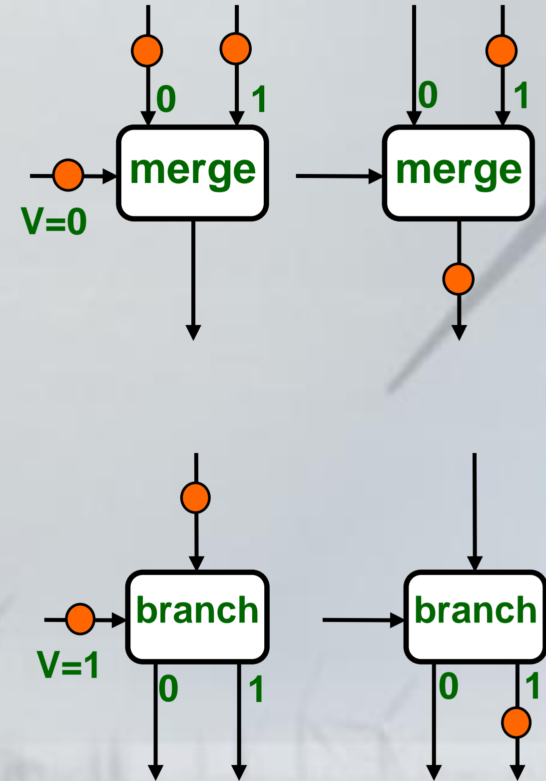
Mapping Function Example

```
function logical_to_linear(logical_addr_t la,
         access_t ac) : linear_addr_t
{
    var seg_sel_t sel;
    var seg_des_t sd;
    var uint<32> des_tab;

    sel = la.$1;
    des_tab = sel.ti==0b0 ? GDTR.base : LDTR.base;

    /* need to look up LDT or GDT first */
    sd = linear_mem[des_tab , la.$1.index];

    /* then call the other translation function */
    return sd.base + la.$2;
}
```

Semantic Model

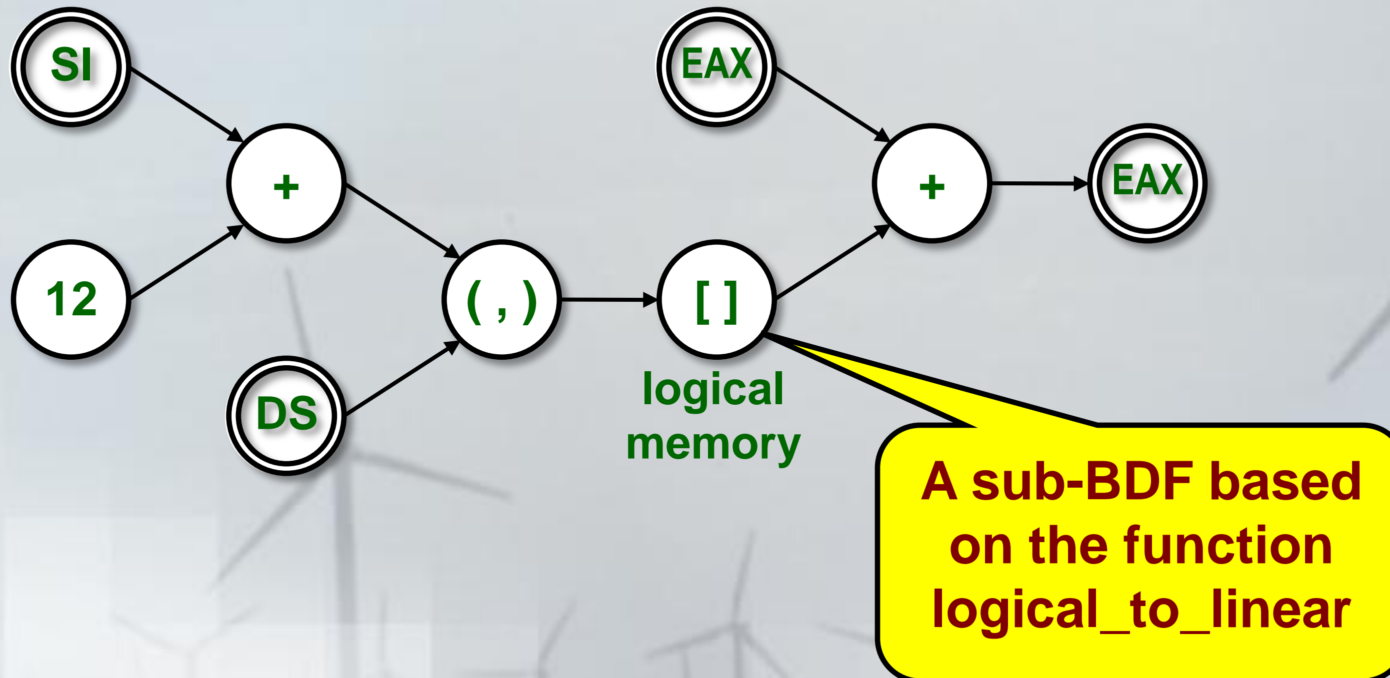
- Boolean Dataflow Model
  - Directed Graph with 4 types of nodes
    - Operator, I/O, merge, branch
  - Valued-tokens flow on edges





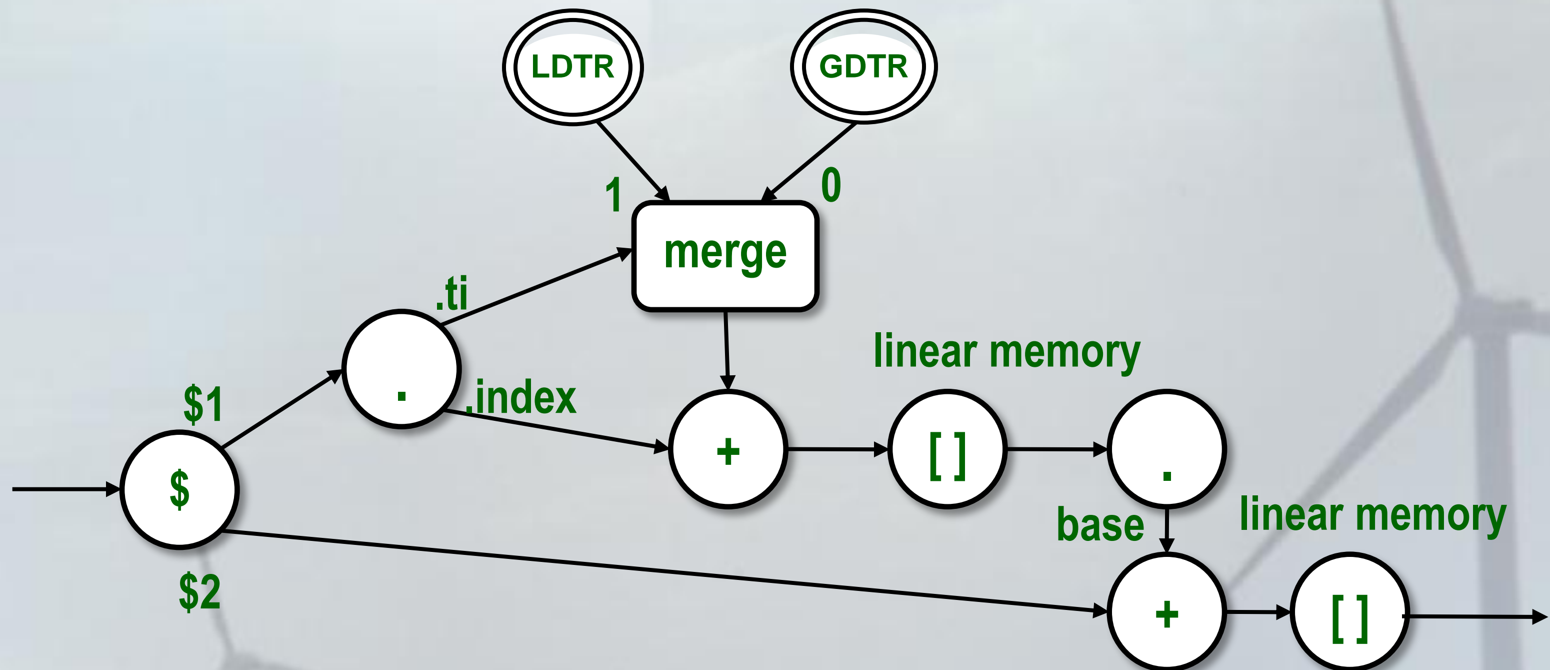
Example BDF

- An Add Instruction
  - EAX = EAX + logical_memory[(DS, SI + 12)]



Example BDF (cont'd)

- The logical_memory subgraph



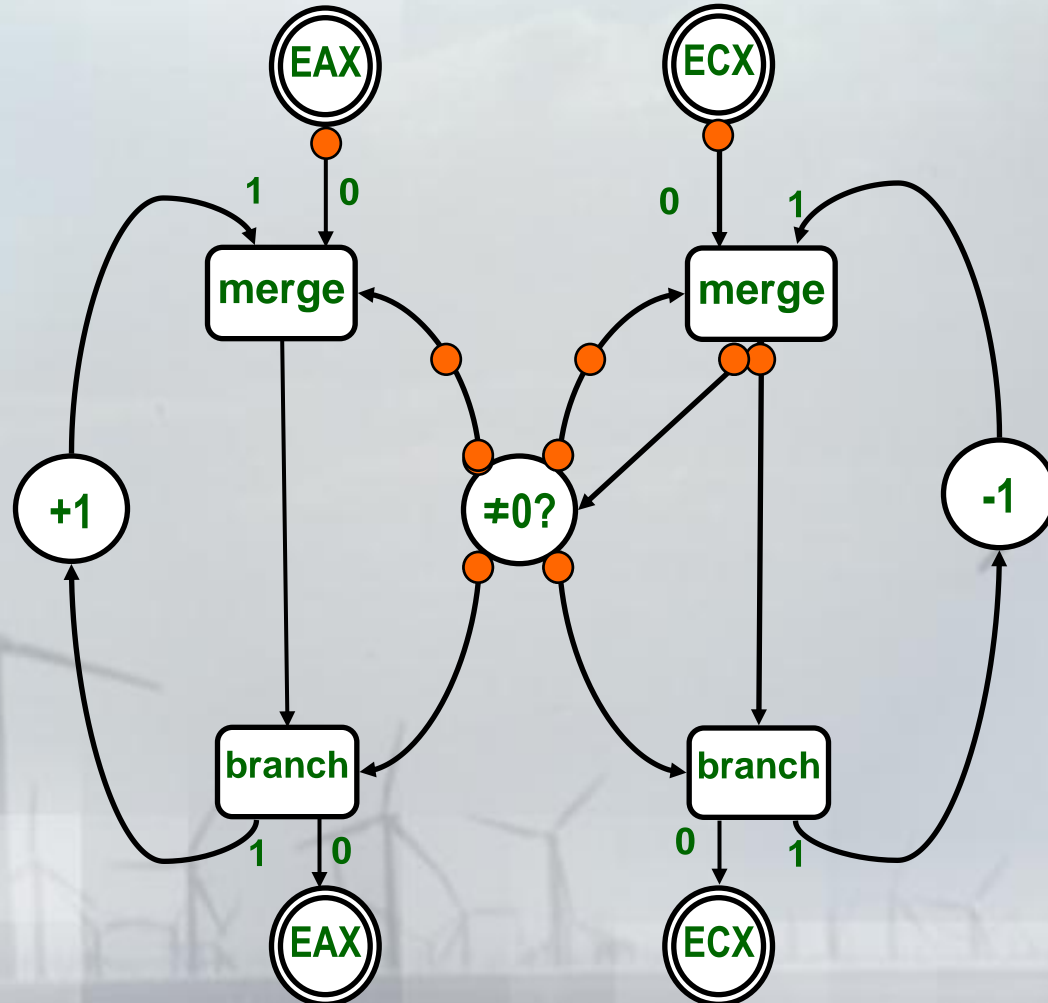

```
sel = la.$1;
des_tab = sel.ti==0b0 ? GDTR.base : LDTR.base;

sd = linear_mem[des_tab, la.$1 .index];
return sd.base + la.$2;
```

Another sub-BDF based on paging

BDF Loop Example

- BDF supports loop, useful for string instructions

```
while (--ECX)
{
  EAX++;
}
```



The two loops can be merged if a tuple type is used.

Comparison of BDF with other Models

- Executable model with well-defined semantics
- More flexible than static DF or dependency graph
  - Edges have two consumption/production rates
  - Supports branch/loop
- More analyzable than KPN or CDFGs
  - Less expressive but sufficient for ISA modeling
- Overall, BDF is adequate for ISA

Exception/Side-effect Modeling

- Raise statement for exception
- Label “side_effect” separates regular statements and exceptions
- Side-effects attached to the main BDF

```
function logical2linear(logical_addr_t la, access_t ac) : linear_addr_t
{
    ……
side_effect:
#if mode==protected
     /* segment not present */
    if (sd.p==0) raise(NP, 0);
    /* write to nonwritable segment */
    if (sd isa code_seg_des_t)  raise(GP, 0);
      ……
#endif
}
```

Applications of ADL

- Assembler/disassembler
  - Encoding and assembly available in the ADL
  - Binary pattern matching ⇒ binary decoder/assembler
  - ASCII pattern matching ⇒ disassembler
- Instruction Set Simulation
  - BDF model is executable
  - Can translate BDF to C for fast simulation
- Documentation generation
- Functional Test Generation
  - Semantics specified in declarative manner
- Formal Verification
  - Equivalence checking
  - Property checking

Conclusion

- An ADL for IA32 is defined
  - A formal computation model
  - New concept of environment
  - Supports memory views
  - Supports exception
- Compiler implemented
  - Parses, type checks, and expands AND/OR trees
  - Output: a list of instruction instances
- Ongoing work
  - Completing the ISA description
  - Refining query APIs for ATG
  - Implementing ISS