University of Glasgow

15 Native code generation

- Characteristics of real machines
- Register allocation
- Intermediate representation
- Code selection

University of Glasgow

Characteristics of real machines

- Code selection is difficult because:
  - CISC machines have very complicated instructions, with multiple addressing modes.
  - even RISC machines have some fairly complicated instructions.
- Register allocation is an issue:
  - Registers should be used as much as possible.
  - RISC machines typically have only general-purpose registers.
  - CISC machines typically have a variety of special-purpose registers (e.g., int registers, float registers, address registers).

University of Glasgow

Register allocation

- Aim to use registers as much as possible for local variables and intermediate results of expressions.
- Problem: The number of registers is limited
  - especially when some are dedicated (e.g., **fp**, **sp**).
- Opportunity: Different variables can be allocated to the same register if they are live at different times.
- Here, a variable is deemed to be **live** only if it might be inspected later.

University of Glasgow

Register allocation: basic-blocks

- A **basic-block** (BB) is a straight-line sequence of instructions:
  - no jumps except at the end of a BB
  - no jumps to anywhere except the start of a BB.
- Within a BB, break up complicated expressions using temporary variables, such that each assignment instruction contains at most one operator. E.g.:

```
a = (b+c)*(d-e);
```

t1 ← b + c
t2 ← d – e
a ← t1 × t2

- *Note:* Basic-blocks are unrelated to block structure.

University of Glasgow

Example: basic-block *(1)*

- Consider the C function:

```
int tri (int a, int b, int c) {
   int s = (a+b+c)/2;
   return s*(s-a)*(s-b)*(s-c);
}
```

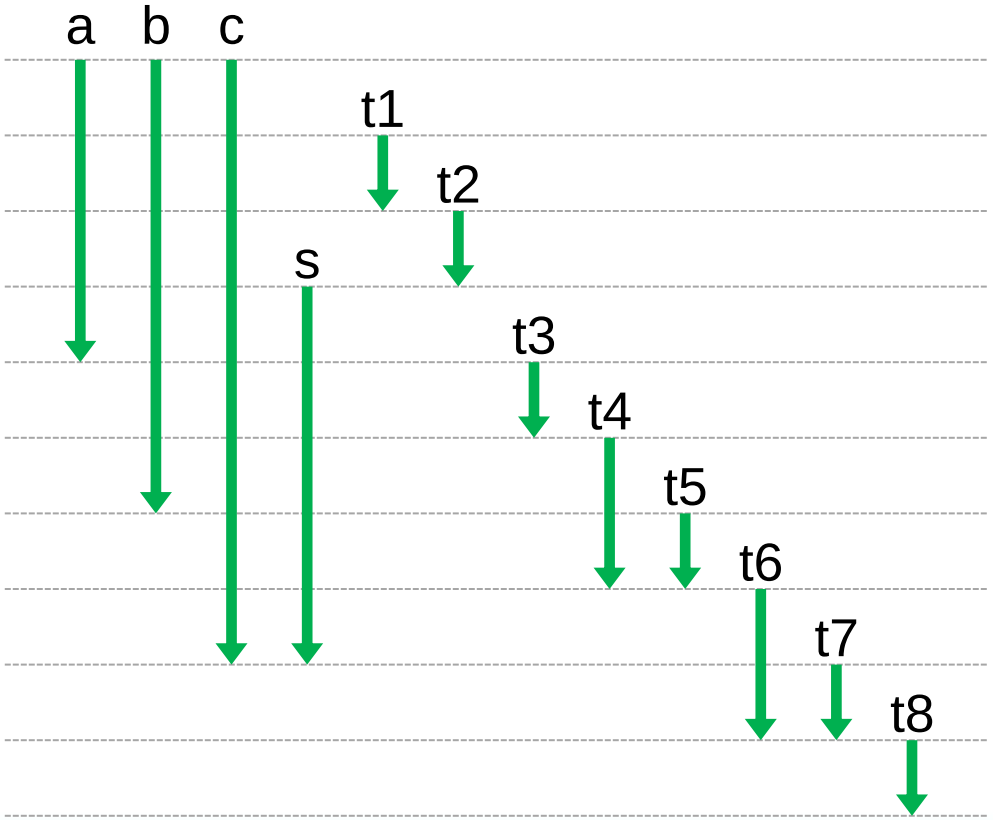
- This function's body is a single BB:

```
t1 ← a + b
t2 ← t1 + c
s ← t2 / 2
t3 ← s – a
t4 ← s × t3
t5 ← s – b
t6 ← t4 × t5
t7 ← s – c
t8 ← t6 × t7
return t8
```

Example: basic-block *(2)*

- Within the BB, determine where each variable is live, then allocate registers:

```
t1 ← a + b
t2 ← t1 + c
s ← t2 / 2
t3 ← s – a
t4 ← s × t3
t5 ← s – b
t6 ← t4 × t5
t7 ← s – c
t8 ← t6 × t7
return t8
```



Allocated: r1 r2 r3 r4 r4 r4 r1 r1 r2 r1 r2 r1

University of Glasgow

Register allocation: control-flow graphs

- A **control-flow graph** is a directed graph in which:
  - each vertex is a BB
  - each edge is a jump from the end of one BB to the start of another BB
  - one vertex is designated as the entry point
  - one vertex is designated as the exit point.

University of Glasgow

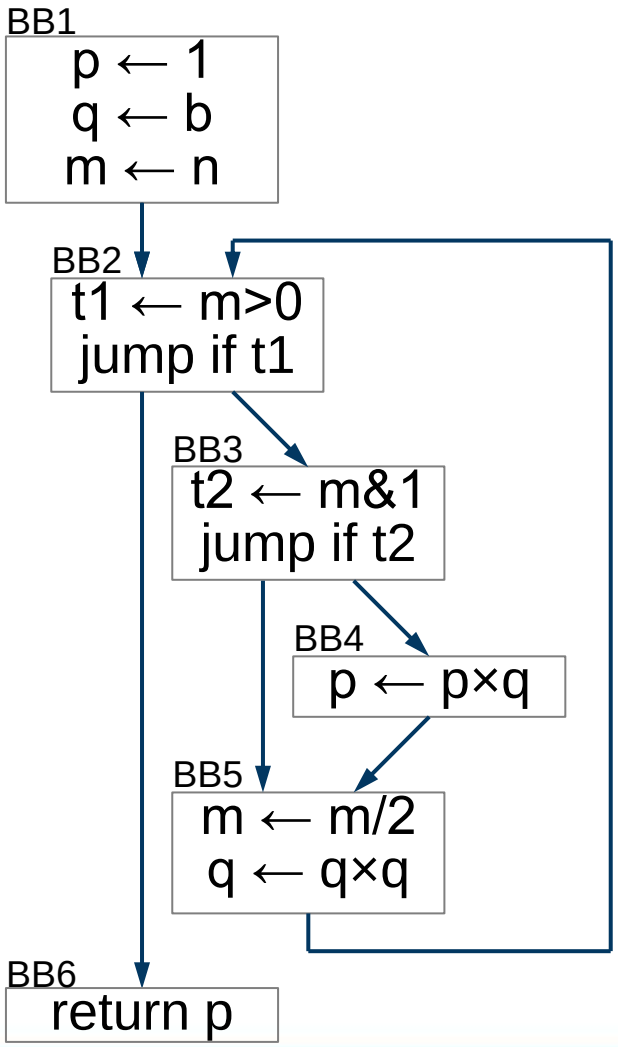
Example: control-flow graph *(1)*

- Consider the C function:

```
int pow (int b, int n) {
   int p = 1, q = b, m = n;
   while (m > 0) {
      if (m & 1) p = p*q;
      m = m/2;
      q = q*q;
   }
   return p;
}
```

Example: control-flow graph *(2)*

- This function's body is a control-flow graph:
- Where is each variable live?
  - b and n are live only in BB1
  - p is live everywhere
  - m and q are live everywhere except in BB6
  - t1 is live only in BB2
  - t2 is live only in BB3.



University of Glasgow

Liveness analysis: data flow equations

- Define the following sets for each BB *b* in a control-flow graph:
  - *in*[*b*] is the set of variables live at the start of *b*
  - *out*[*b*] is the set of variables live at the end of *b*
  - *use*[*b*] is the set of variables *v* such that *b* inspects *v* (before any update to *v*)
  - *def*[*b*] is the set of variables *v* such that *b* updates *v* (before any inspection of *v*)
- Data flow equations for liveness analysis:

  $in[b] = use[b] \cup (out[b] - def[b])$

  $out[b] = in[b'] \cup in[b''] \cup \ldots$
  (where b', b'', … are the successors of *b* in the flow graph)

University of Glasgow

Liveness analysis: algorithm

- The liveness analysis algorithm follows directly from the data flow equations:

  To compute in[*b*] and out[*b*] for all BBs in a control-flow graph:

  1. For each *b*:
     1.1. Set *in*[*b*] = *out*[*b*] = { }.
  2. Repeat until the sets *in*[*b*] and *out*[*b*] stop changing:
     2.1. For each *b*:
        2.1.1. Set *in*[*b*] = *use*[*b*] U (*out*[*b*] – *def*[*b*]).
        2.1.2. Set *out*[*b*] = *in*[*b'*] U *in*[*b''*] U …
        (where *b'*, *b''*, … are the successors of *b*).

University of Glasgow

Intermediate representation

- Native code generation is simplified by using a low-level **intermediate representation** (**IR**) of the source program.
- The IR should be capable of:
  - representing the semantics of the source code
  - representing the semantics of target-machine instructions.
- The IR should ideally be independent of the target machine.

University of Glasgow

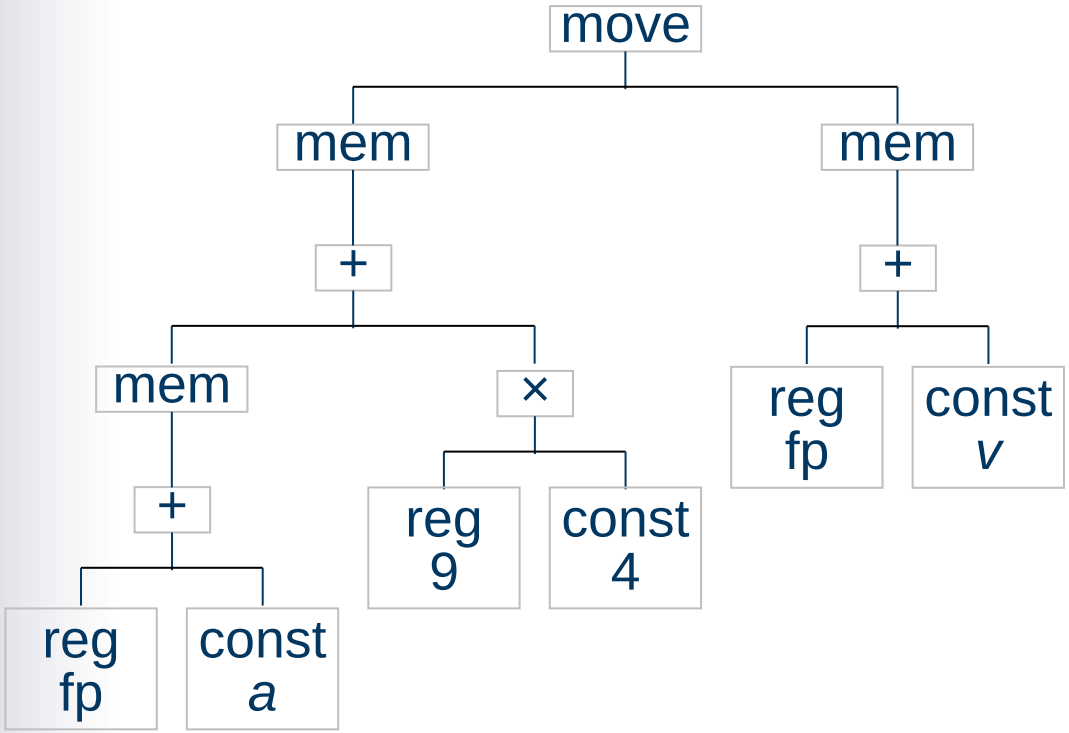
Example: IR tree *(1)*

- Consider the C assignment "`a[i] = v;`".
- Assume that:
  - `a` has type `int*` and `v` has type `int`
  - each `int` occupies 4 bytes
  - variable `a` is located at offset *a* within the topmost activation frame (that location contains the base address of `a`)
  - variable `i` is located in register r9
  - variable `v` is located at offset *v* within the topmost activation frame.
- Address of `a[i]` is:

  (base address of `a`) + 4×(content of `i`).

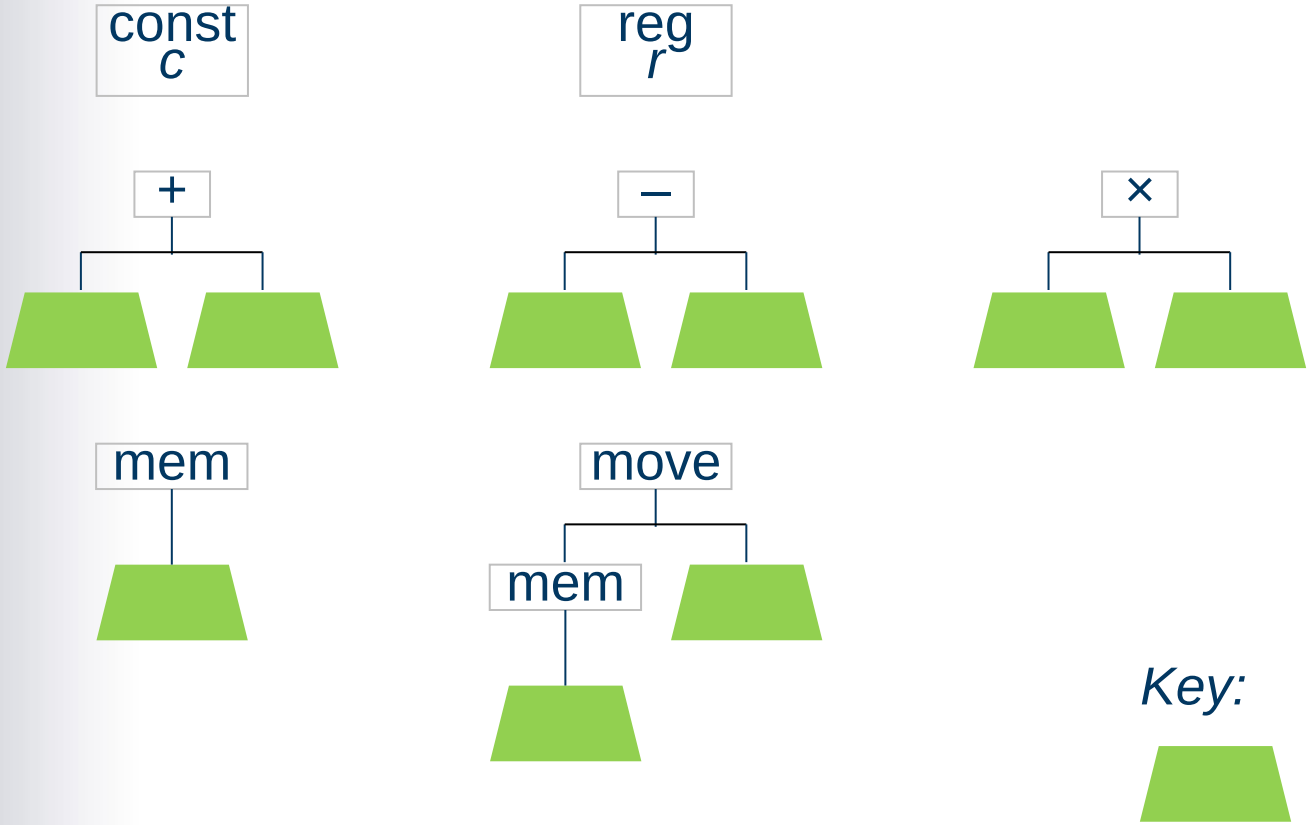
Example: IR tree *(2)*

- Possible IR tree for "`a[i] = v;`":



Summary of IR trees

- IR trees:



University of Glasgow

Modelling target instructions

- Model the semantics of each target machine instruction using an IR tree pattern.
- *Note:* We use the same IR to model both source code and target instructions.

University of Glasgow

Example: Jouette *(1)*

- **Jouette** is a hypothetical RISC machine
  - invented by Andrew Appel for his *Modern Compiler Implementation* books.
- Jouette architecture:
  - general-purpose registers r0, r1, r2, ..., r31
  - r0 always contains zero.

Example: Jouette *(2)*

- Jouette instruction set:

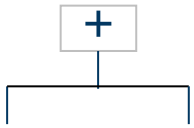
Mnemonic	*Behaviour*
`ADD` *r*,*r'*,*r''*	$r \leftarrow r' + r''$
`SUB` *r*,*r'*,*r''*	$r \leftarrow r' - r''$
`MUL` *r*,*r'*,*r''*	$r \leftarrow r' \times r''$
`ADDI` *r*,*r'*,*c*	$r \leftarrow r' + c$
`SUBI` *r*,*r'*,*c*	$r \leftarrow r' - c$
`LOAD` *r*,*c*(*r'*)	$r \leftarrow \text{mem}[r' + c]$
`STORE` *r*,*c*(*r'*)	$\text{mem}[r' + c] \leftarrow r$
`COPY` (*r*),(*r'*)	$\text{mem}[r'] \leftarrow \text{mem}[r]$

r, *r'*, *r''* are registers; *c* is a constant

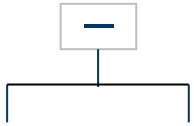
Example: modelling Jouette instructions *(1)*

- Jouette arithmetic instructions:

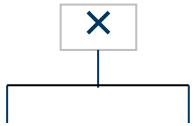
`ADD` *r*,*r'*,*r''*
($r \leftarrow r' + r''$)



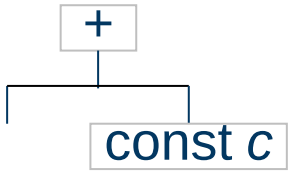
`SUB` *r*,*r'*,*r''*
($r \leftarrow r' - r''$)

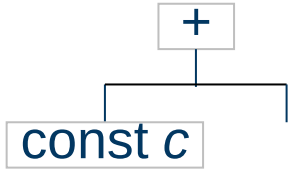


`MUL` *r*,*r'*,*r''*
($r \leftarrow r' \times r''$)



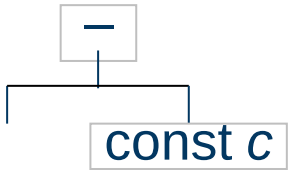
`ADDI` *r*,*r'*,*c*
($r \leftarrow r' + c$)





const *c*
(*r'* = r0)

`SUBI` *r*,*r'*,*c*
($r \leftarrow r' - c$)

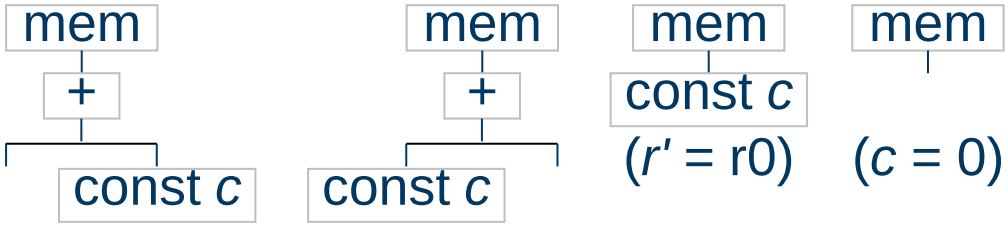


It's possible for >1 patterns to model 1 instruction.

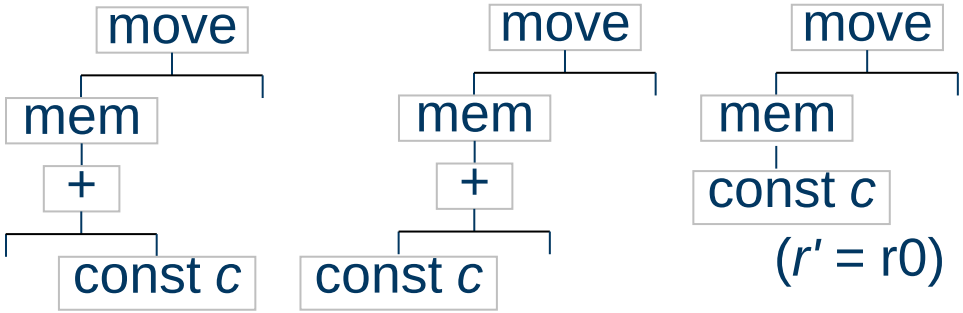
Example: modelling Jouette instructions *(2)*

- Jouette load/store instructions:

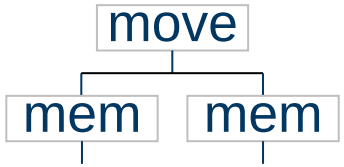
`LOAD` *r*`,`*c*`(`*r′*`)`
($r \leftarrow \text{mem}[r' + c]$)



`STORE` *r*`,`*c*`(`*r′*`)`
($\text{mem}[r' + c] \leftarrow r$)



`COPY` `(`*r*`),(`*r′*`)`
($\text{mem}[r'] \leftarrow \text{mem}[r]$)



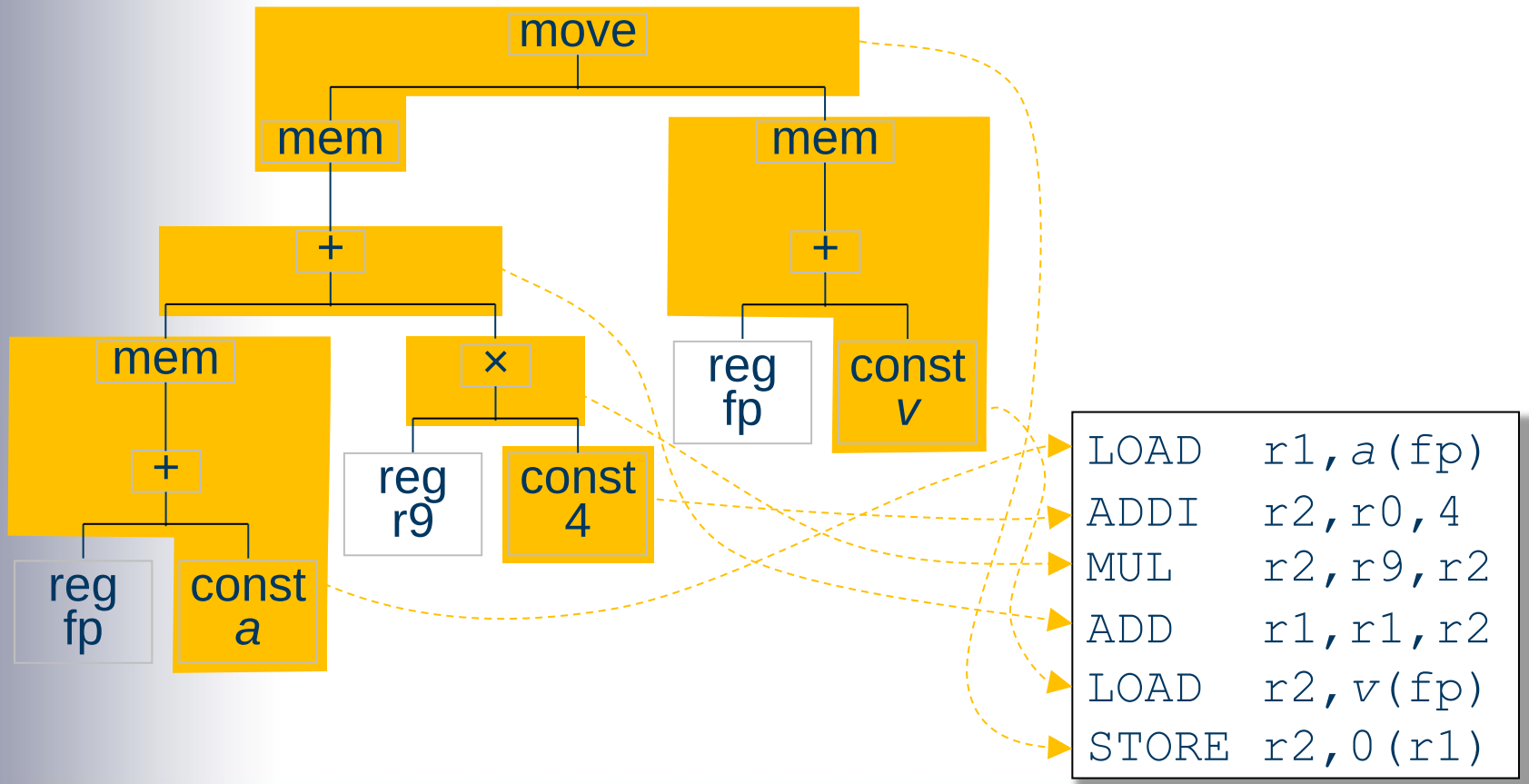
University of Glasgow

Code selection: method

- Translate the source code or AST into an IR tree.
- “Cover” the tree with IR instruction patterns.
- Emit code corresponding to these instruction patterns
  - performing register allocation as you go.

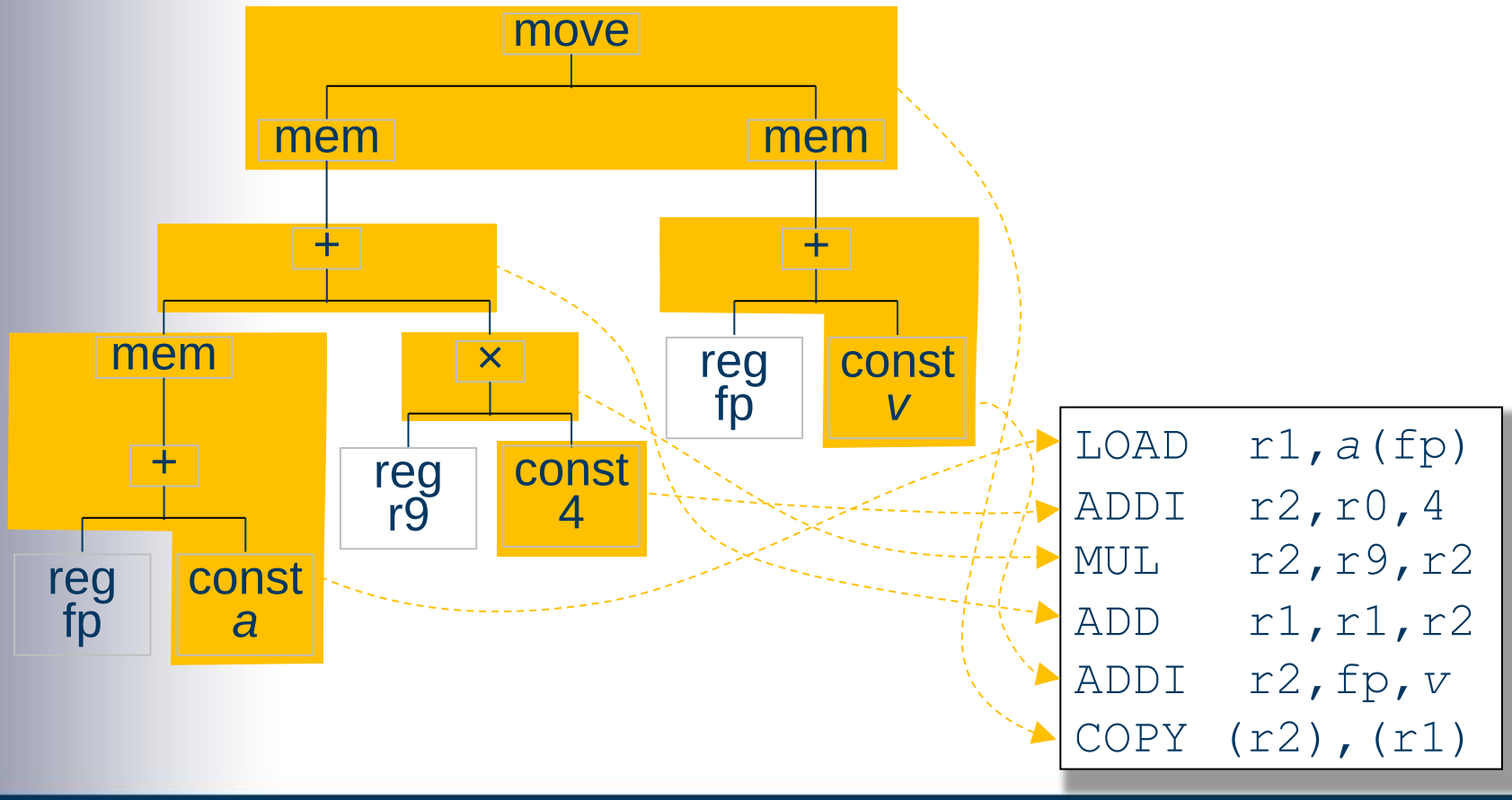
Example: code selection *(1)*

- One way to cover the IR for "`a[i] = v;`":



Example: code selection *(2)*

- Different way to cover the IR for "`a[i] = v;`":



University of Glasgow

Code selection: maximal-munch algorithm

- Maximal-munch code selection algorithm:

  To cover the IR tree *t* using instruction patterns *ps*:

  1. Find the largest pattern *p* in *ps* that covers the top of *t*.
  2. For each uncovered subtree *s* of *t* (from left to right):
     2.1. Cover *s* using *ps*.
     2.2. Emit the instruction corresponding to *p*.

- The time complexity is O(size of *t*).
- The emitted code is optimal in the sense that:
  - no two adjacent patterns could be replaced by a single pattern
  - the number of instructions is minimal.