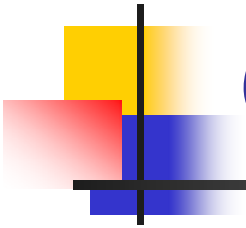
An Architecture-Oriented Approach to Code Compression

Cheng-Hong Li
Steven M. Nowick
Columbia University



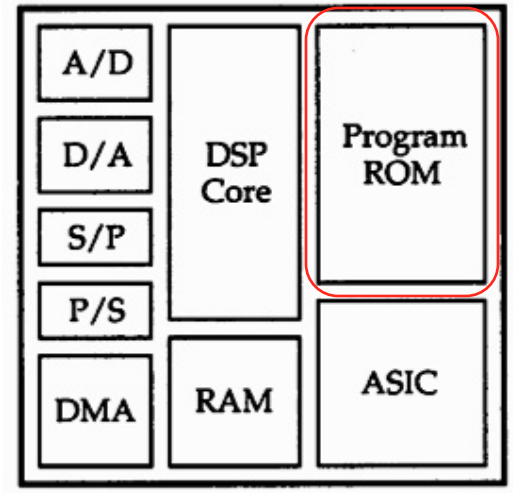
Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

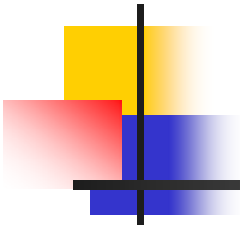
Motivation

- Embedded systems usually have limited storage space for programs
  - Cost, system size, and power constraints

[Liao, Devadas, Keutzer99]



- In many cases, a significant amount area is devoted to program ROM.
- If program size can be reduced, one may:
  - Squeeze in more functionality, OR
  - Reduce area and power



Some Existing Solutions

- Denser instruction sets
  - Ex: Thumb (16-bit ISA for ARM): 70% cmp. ratios

  + Supported by processors

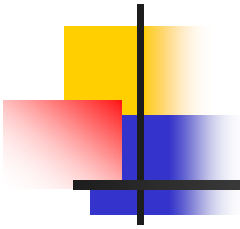
  - Requires new processors & new compilers
- Compiler optimizations for smaller footprints
  - Limited improvement: >70% cmp. ratios [Debray et al. 00]
- Applying compression techniques - “Code compression”
  - Ex: IBM CodePack for PowerPC [Kemp et al.98]
  - 60% cmp. ratios

  + *(Sometimes)* Can use existing processors

  + *(sometimes)* Can use existing compilers

  - Requires added hardware decoder

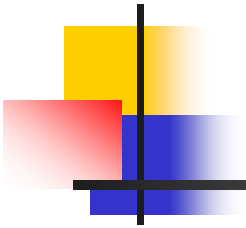
⇨ *Our focus*



Solutions - Code Compression

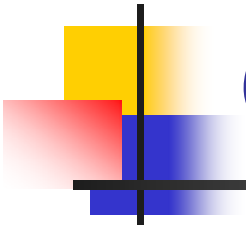
- Reduce program size using data compression
  - Compress programs off-line
  - Store compressed programs in memory
  - When executing, decompress *on the fly* by a hardware decoder
- Challenge:
  - On-line decoding
  - Limited hardware resource
  - Everything counts!

⇨ Many existing compression techniques becomes ineffective.



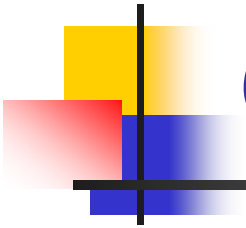
Background

- Statistical coding
  - More frequent symbols receive shorter codewords.
    - Ex: Huffman coding
- “Skew” frequency distributions
  - Combine highly-correlated input units to form new symbols.
  - Modeling: arrange symbols to a number of symbol sets
  - Both techniques are exploited in this paper.



Our Contributions

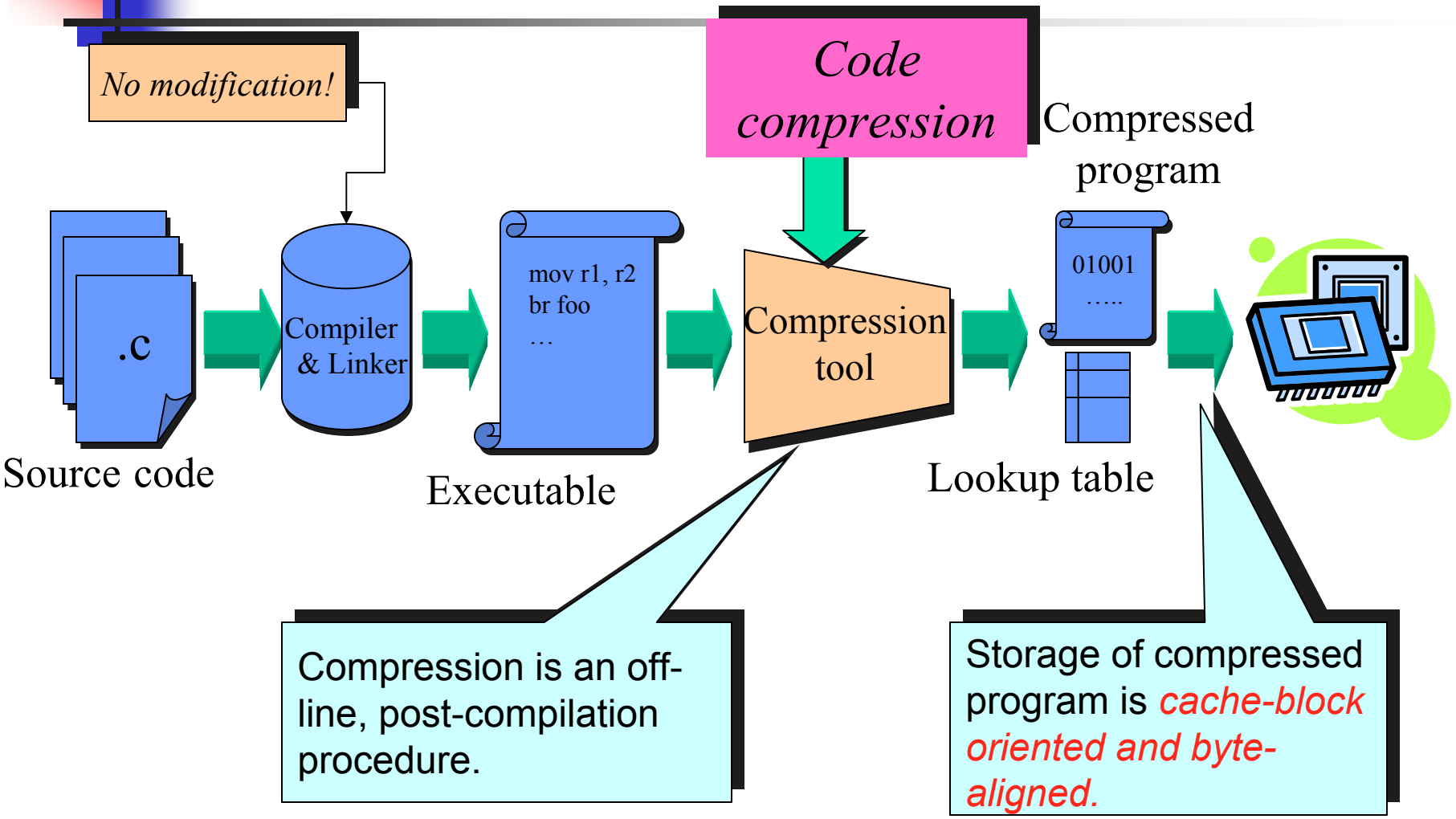
- Exploit higher-order entropies
  - Use architecture-oriented context & state models
- Architecture-based symbol formation
  - “Field-oriented” symbols: tailored to instruction fields
- New Huffman-based coding technique
  - “Curve-fitting”: use single optimal “shared” code
  - Avoids overhead of multiple Huffman codes
- Simple and extensible hardware decoder design



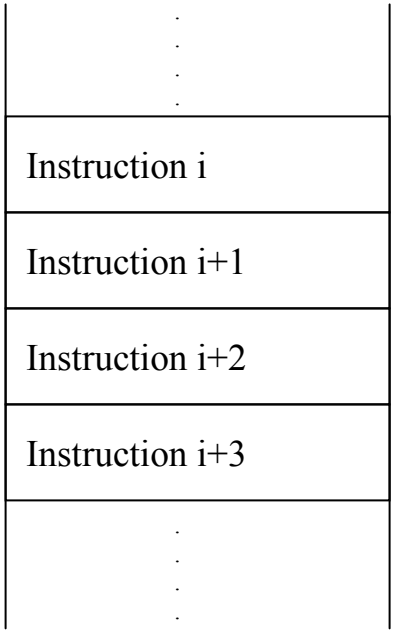
Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

Overview - Compression (1)

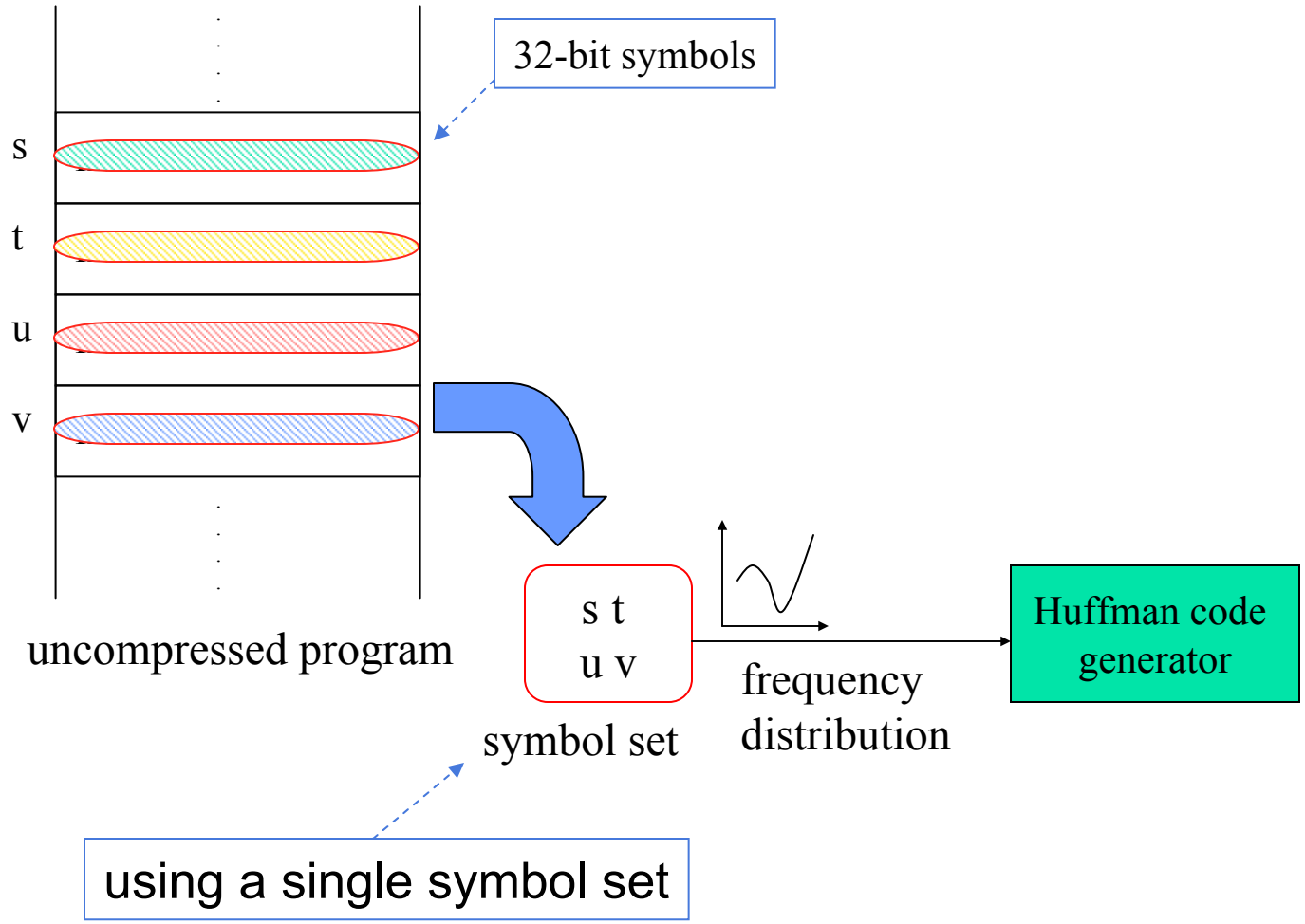


Overview - Compression (2)

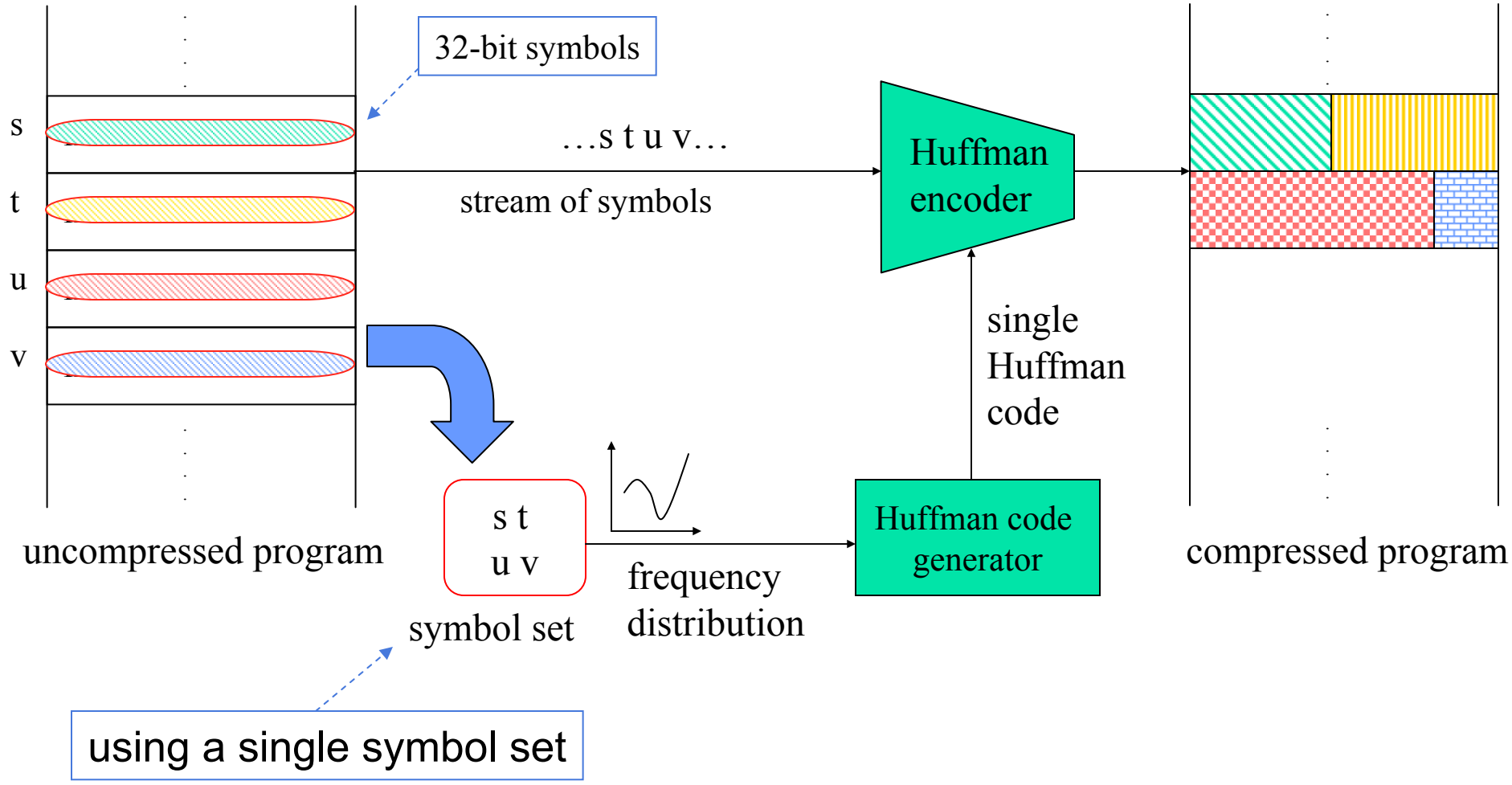


uncompressed program

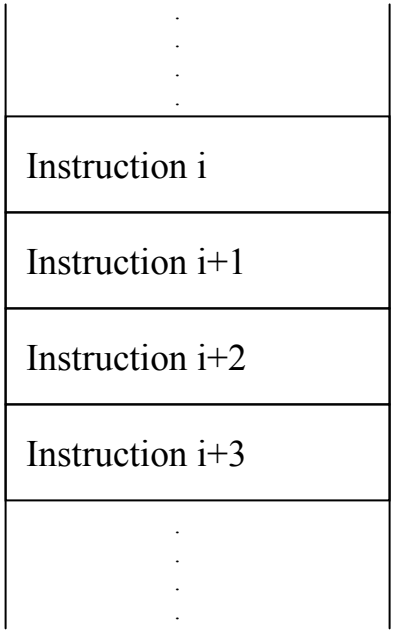
Overview - Compression (2)



Overview - Compression (2)

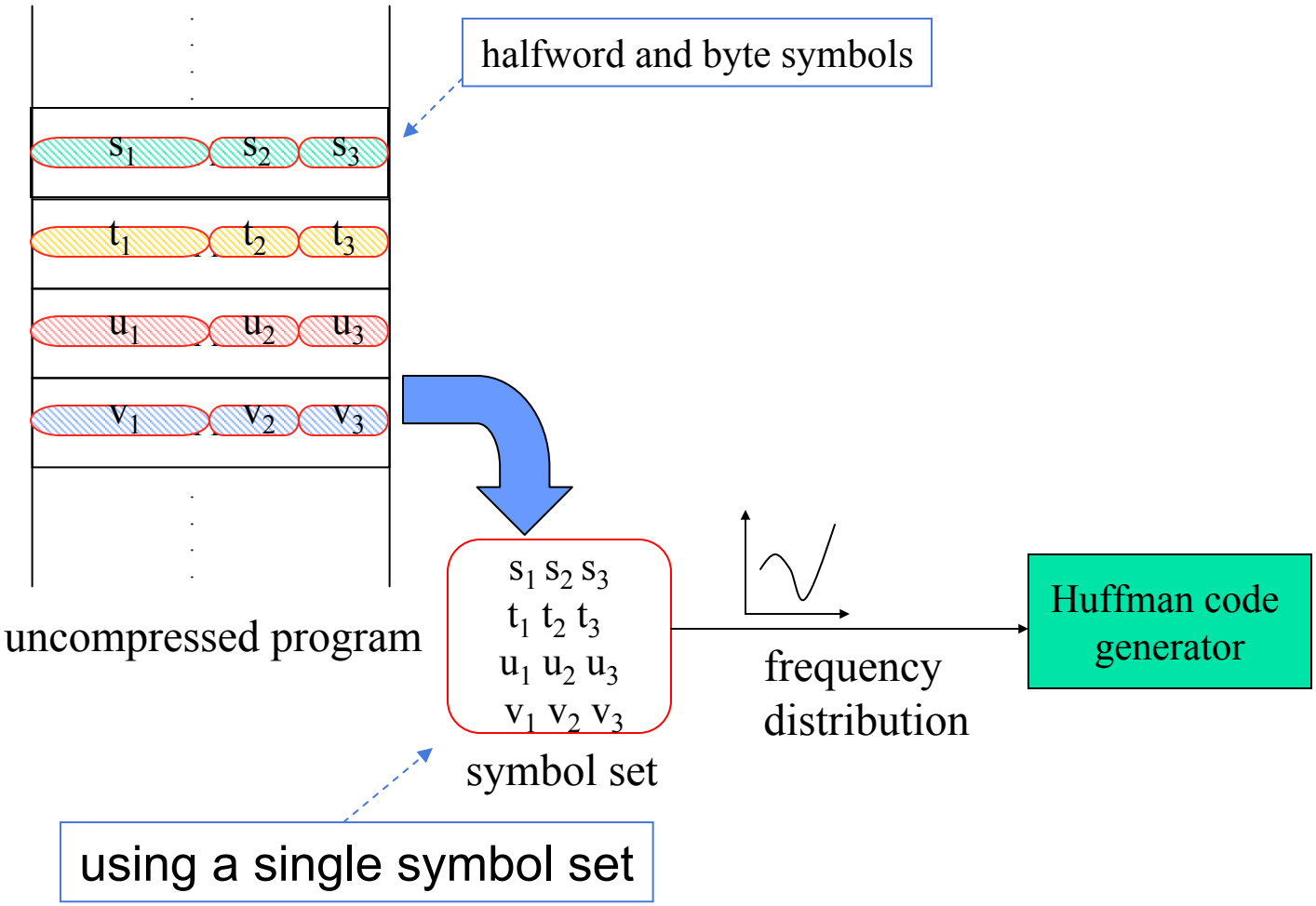


Overview - Compression (3)

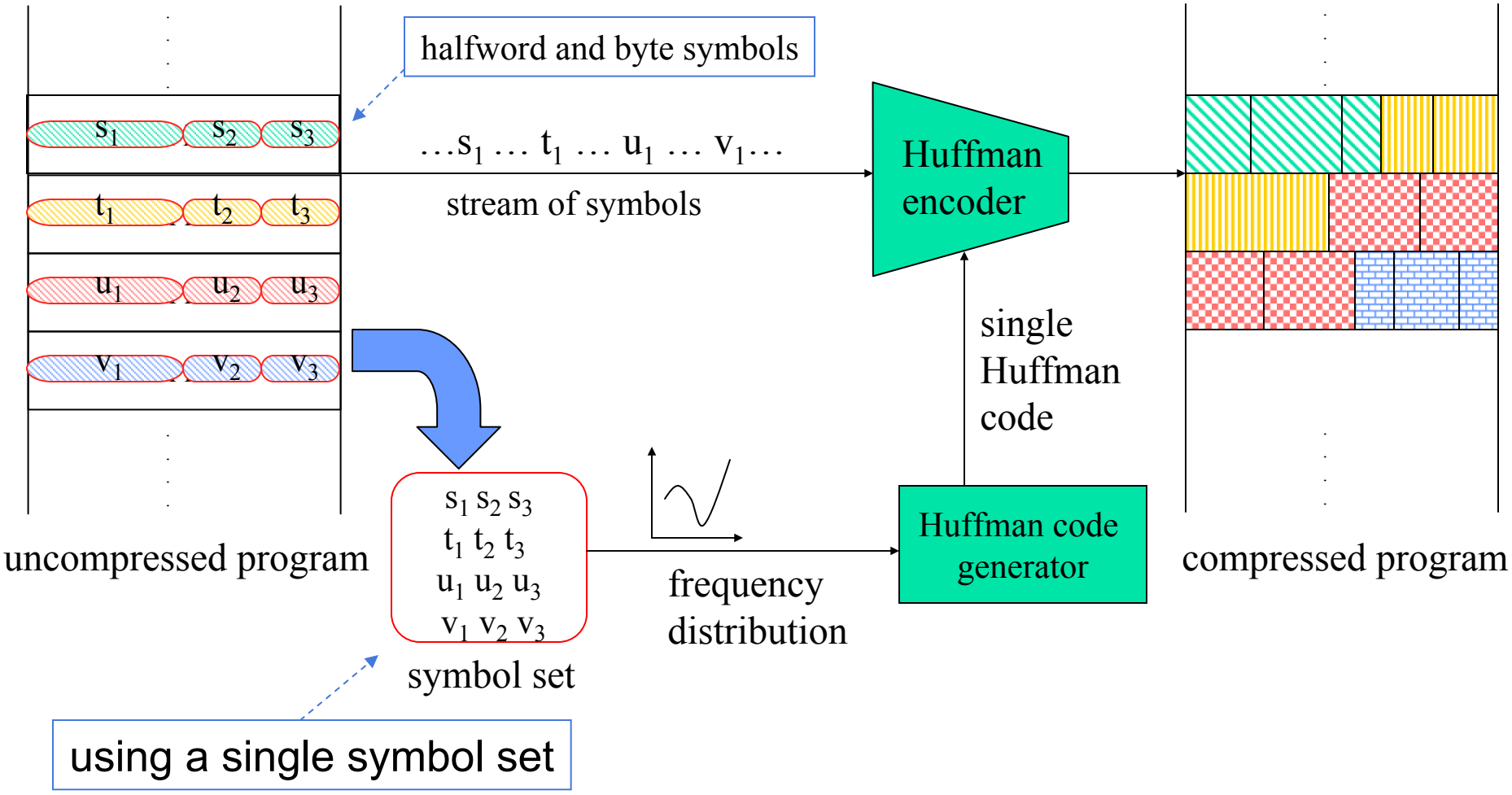


uncompressed program

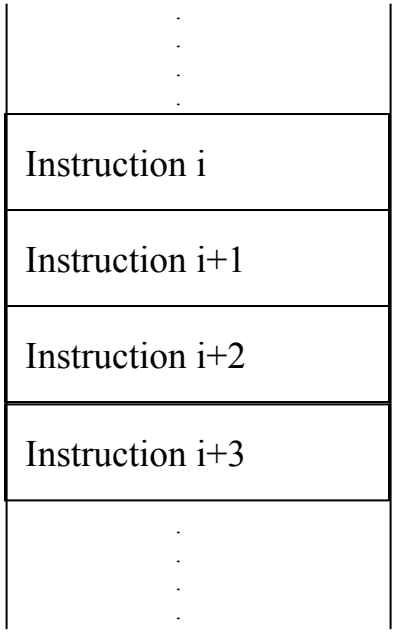
Overview - Compression (3)



Overview - Compression (3)

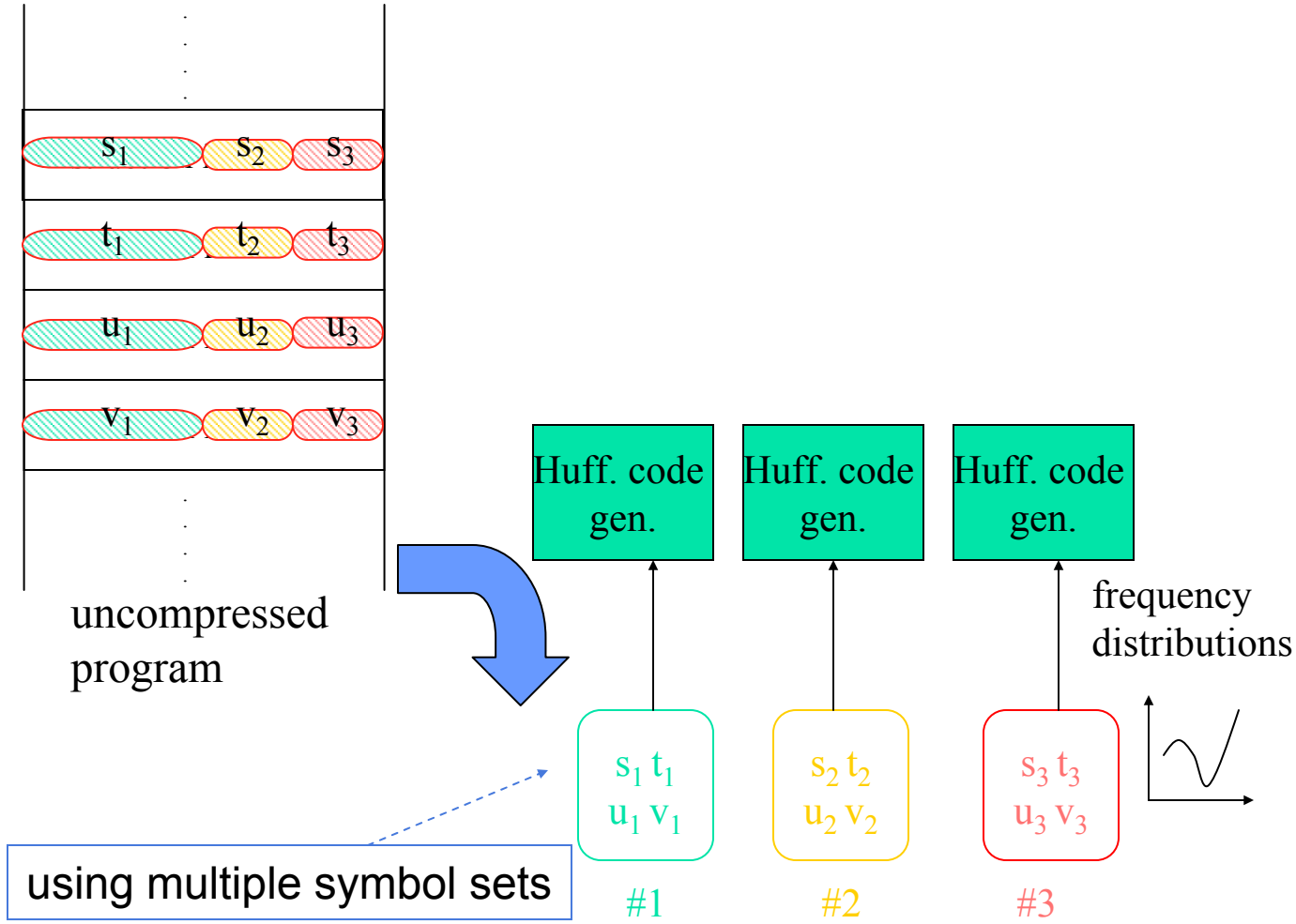


Overview - Compression (4)

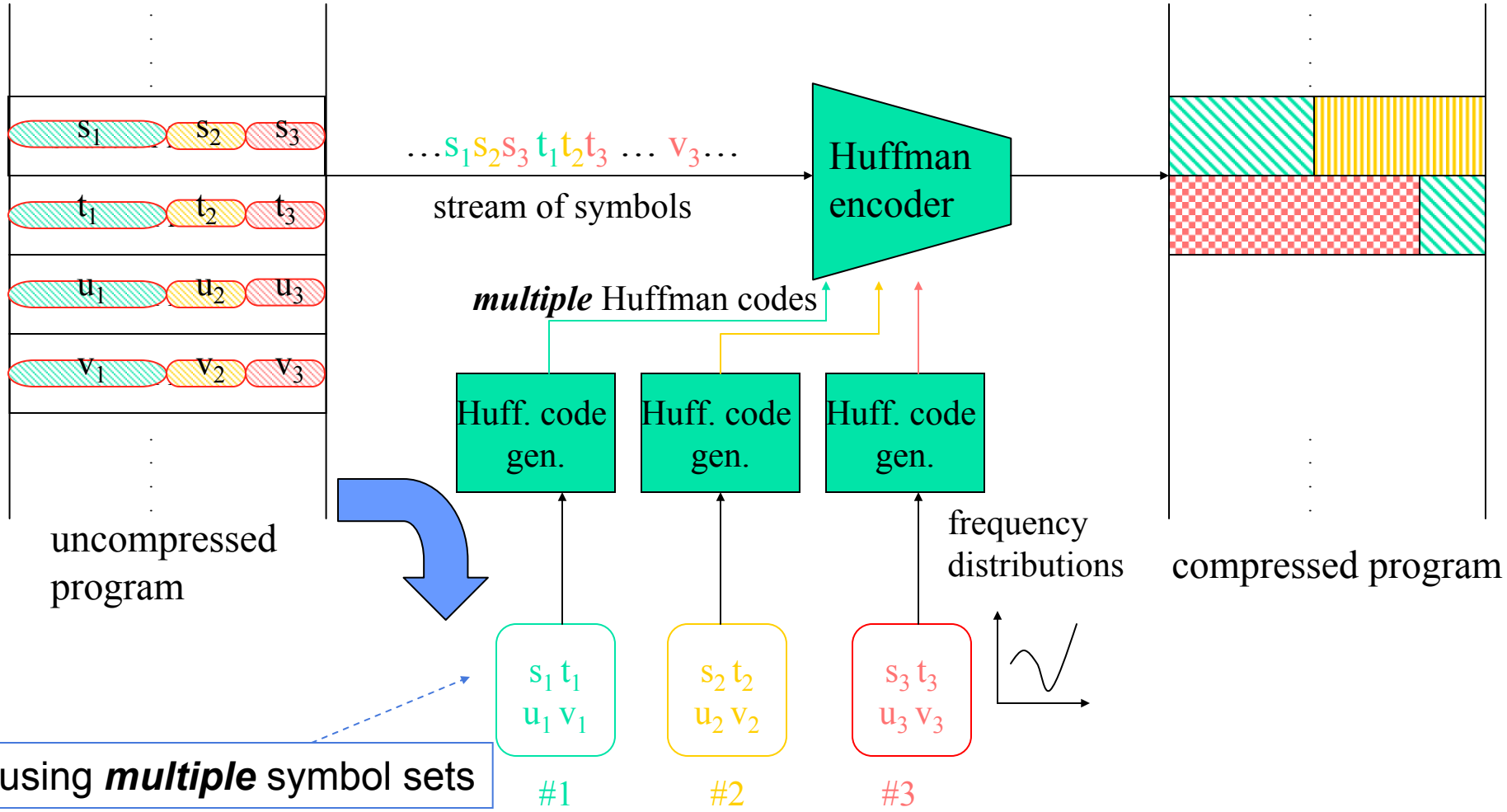


uncompressed program

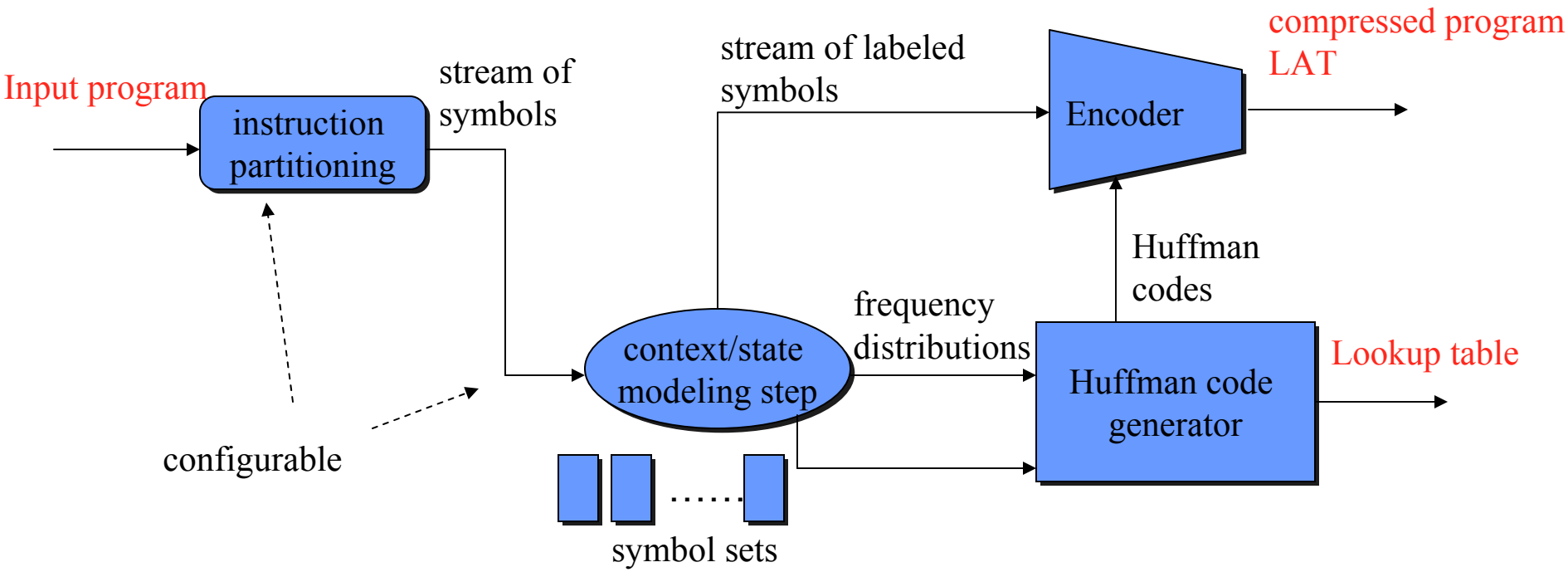
Overview - Compression (4)



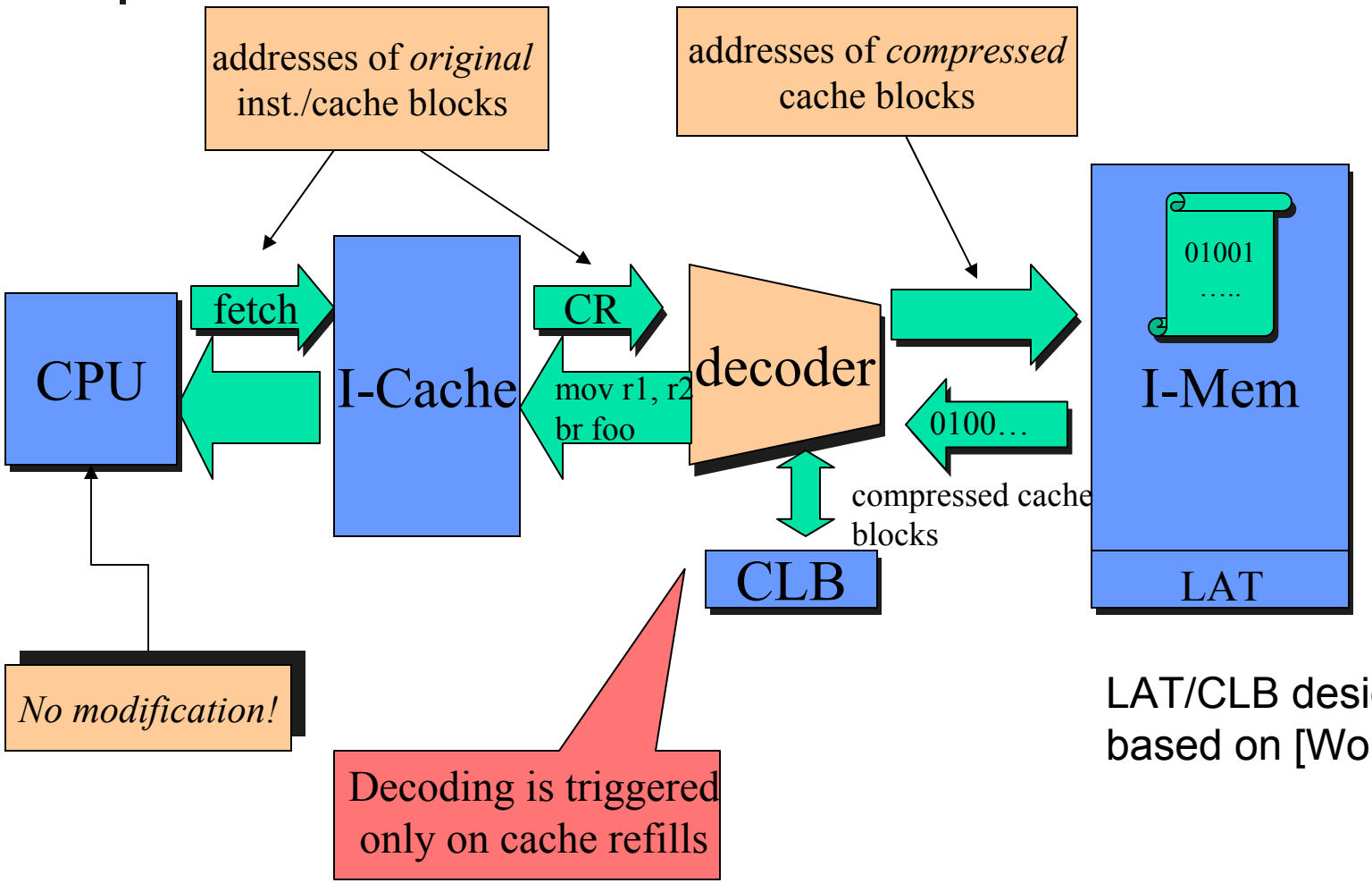
Overview - Compression (4)



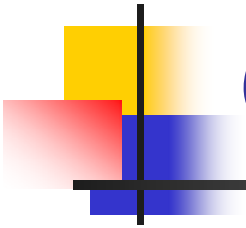
Compression Flow



Overview - Decompression



LAT/CLB designs are based on [Wolfe92].



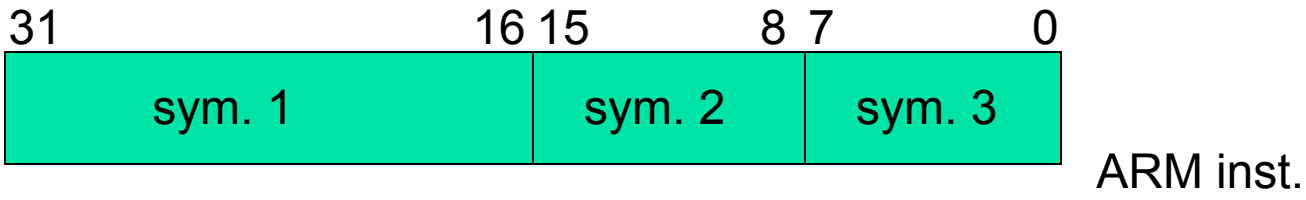
Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

Symbol Formation and Instruction Partitioning (1)

- Symbol: an atomic compression unit
- Three types of symbols:
  - Byte, halfword

Ex: partitioning an ARM inst. to one halfword and two bytes.



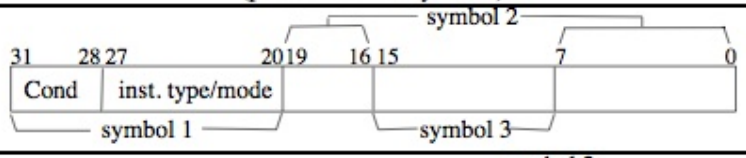
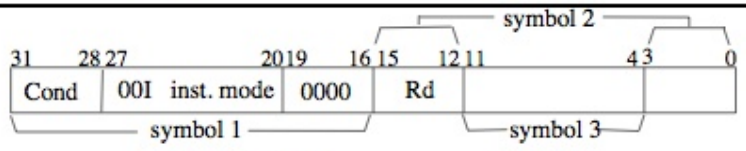
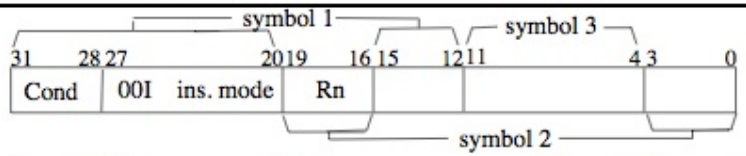
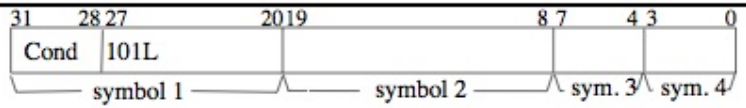
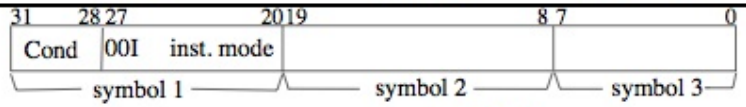
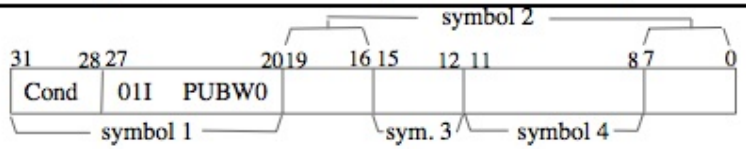
  - Field-oriented
    - Consists of a number of adjacent/non-adjacent fields (variable length)

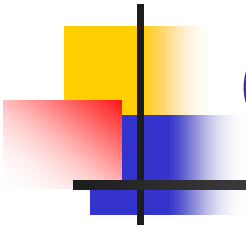
Symbol Formation and Instruction Partitioning (2)

- Instruction classes:

itype classes	**imode** classes
Data proc. (T1)	`tst/teq/cmp/cmn` (I bit not set) (M1)
	`mov/mvn` (I bit not set) (M2)
	Other data proc. (I bit not set) (M3)
	`tst/teq/cmp/cmn` (I bit set)(M4)
	`mov/mvn` (I bit set) (M5)
	Other data proc. (I bit set) (M6)
Data trans. (T2)	`ldr` (M7)
	`str` (M8)
Branch (T3)	Forward branch with L bit set (M9)
	Forward branch with L bit not set (M10)
	Backward branch with L bit set (M11)
	Backward branch with L bit not set (M12)
Misc. (T4)	Misc. (M13)

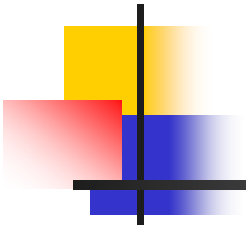
Symbol Formation and Instruction Partitioning (3)

Instruction layouts (partitioned into symbols)	Inst. classes (itype & imode)	Examples
	T2, T4, M7, M13	ldr r3, [r6, #120] s1 s2 s2 1111 0101 1001 0110 0011 0000 0111 1000 s3
	M2, M5	mov r2, r0 t1 t2 t2 1111 00011010 0000 0010 0000 0000 0000 t3
	T1, M1, M4	cmp r2, r3 u1 u1 u3 1111 00010101 0010 0000 0000 0000 0011 u2 u2
	T3,M9-M12	beq <L1> 0000 10100000 0000 0000 0000 0000 000 v1 v2 v3 v4
	M3, M6	add r1, r1, #1 1111 00101000 0001 0001 0000 0000 0001 w1 w2 w3
	M8	str r1, [r3] x1 x2 x2 1111 01011000 0011 0001 0000 0000 0000 x3 x4



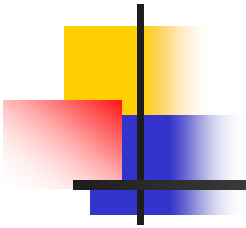
Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work



Modeling

- Modeling: arranging symbols to symbol sets
- *Goal: to skew the freq. distributions*
  - Lower entropy ➔ better compression ratio
- Three types of models are proposed:
  - Context: previous symbols
  - State: current symbol position
  - Hybrid: context and state

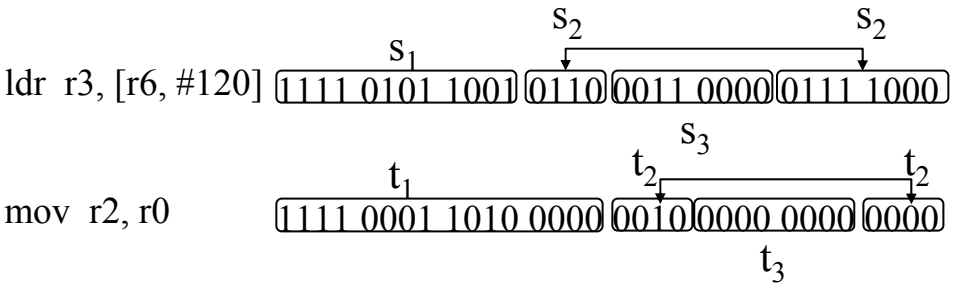


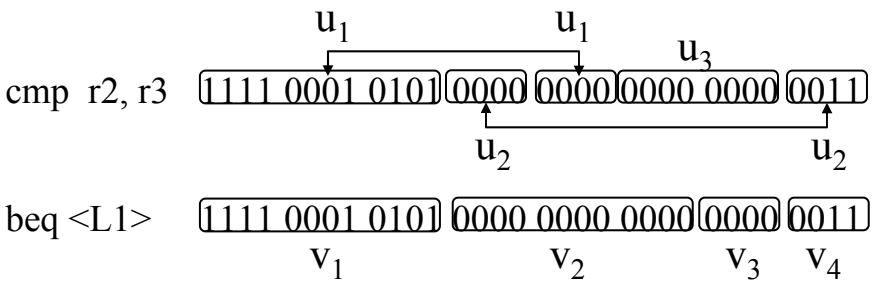
Modeling - Context Model (1)

- Context model: using *instruction class* as context
- For each instruction:
  - The first symbol provides the context (type/mode).
  - The remaining symbols are deposited into a dedicated symbol set.
  - The first symbol is treated as having *no context.*
- Two options:
  - *itype* (4 classes) or *imode* (13 classes)

Modeling - Context Model (2)

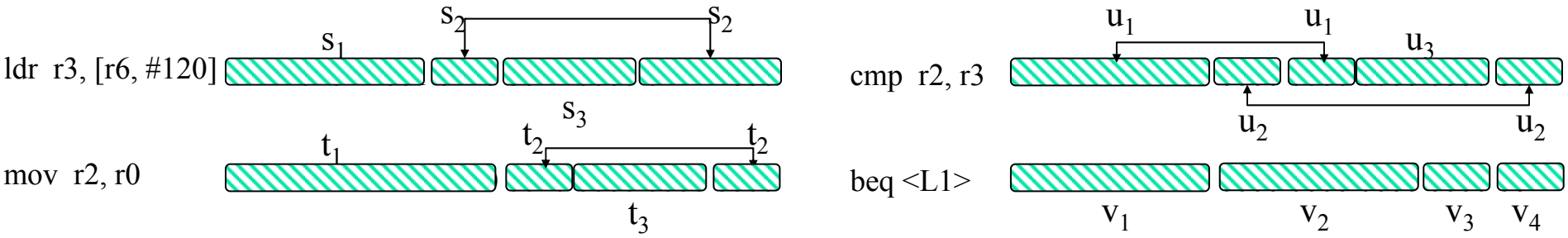
itype model example:





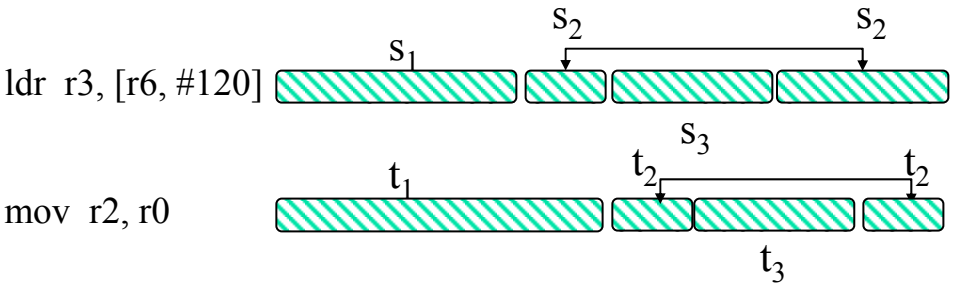
Modeling - Context Model (2)

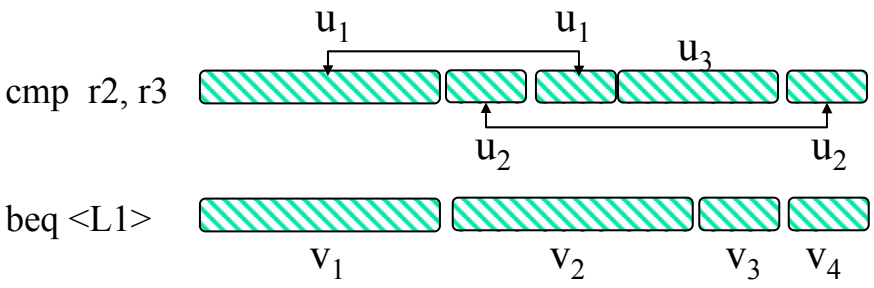
itype model example:

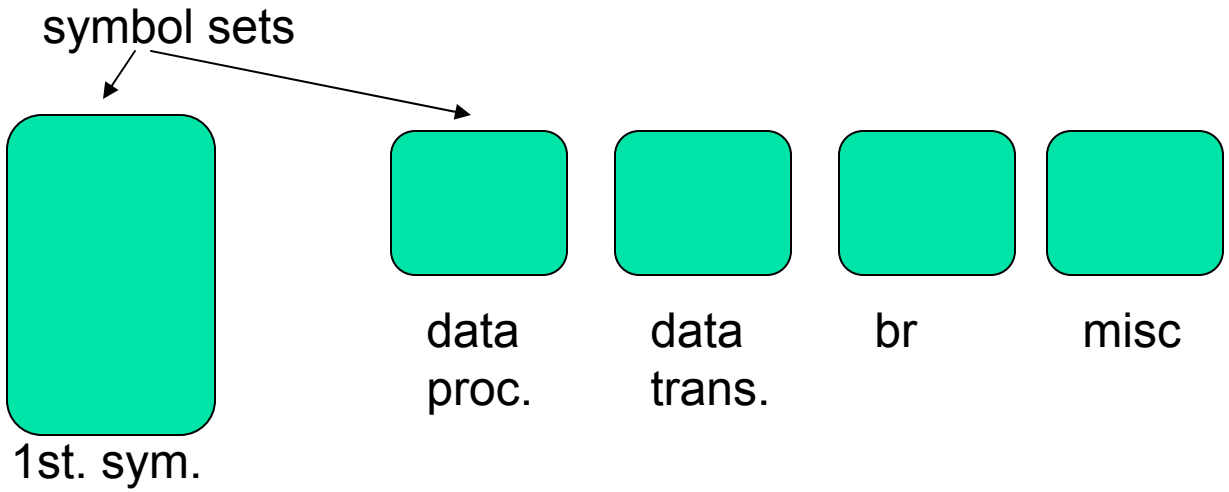


Modeling - Context Model (2)

itype model example:

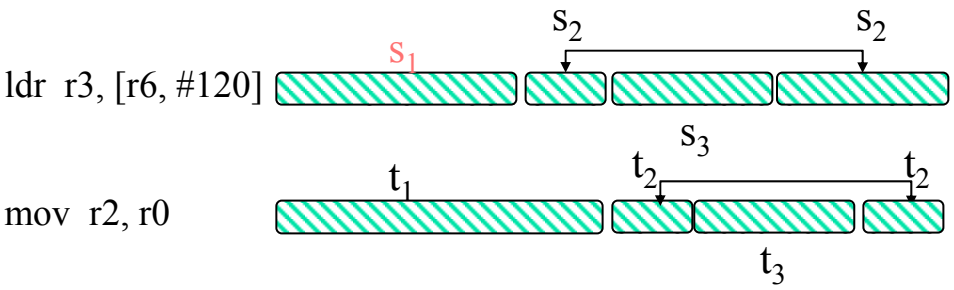


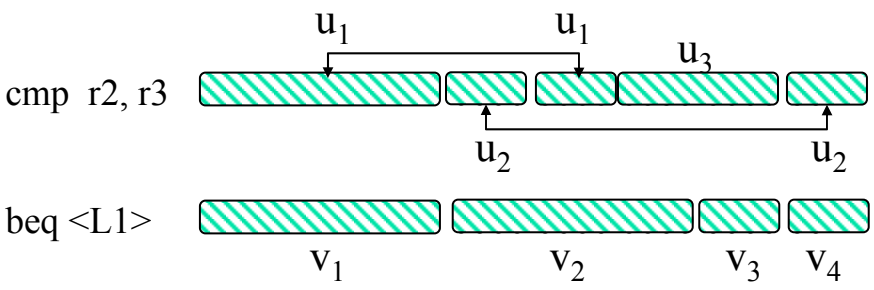


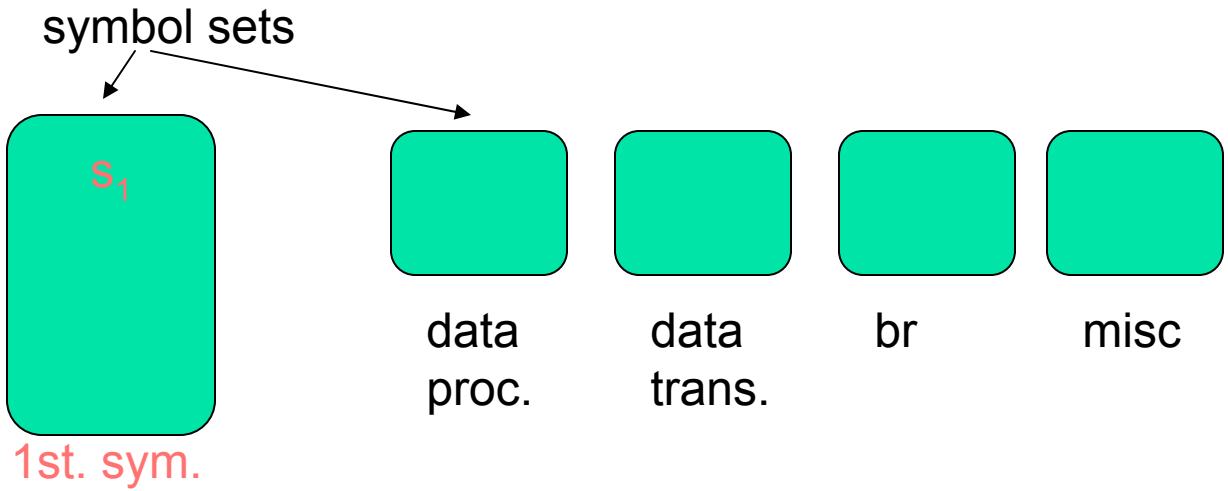


Modeling - Context Model (2)

itype model example:

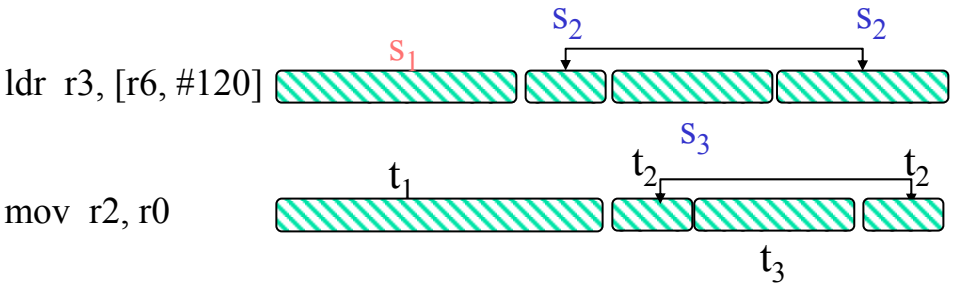


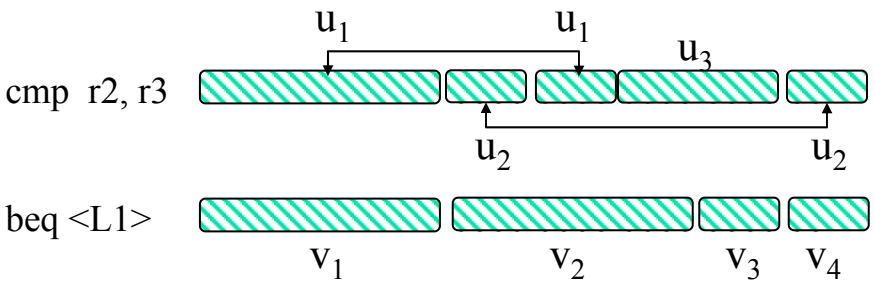


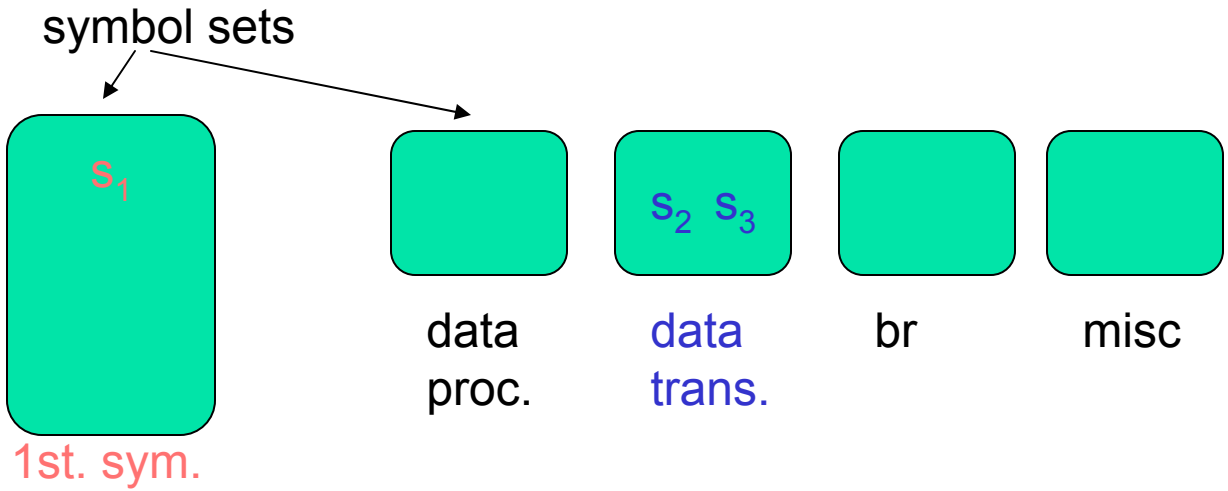


Modeling - Context Model (2)

itype model example:

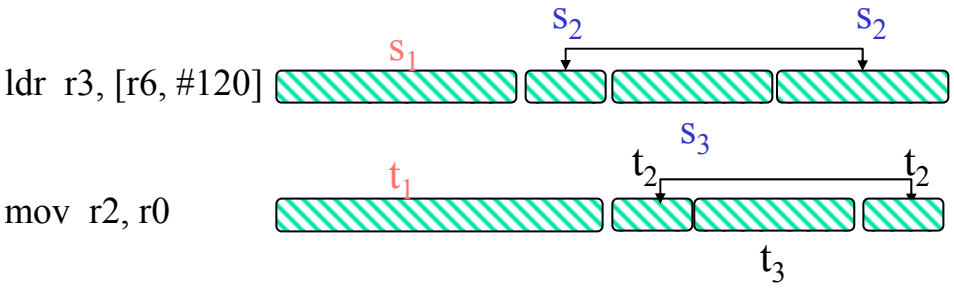


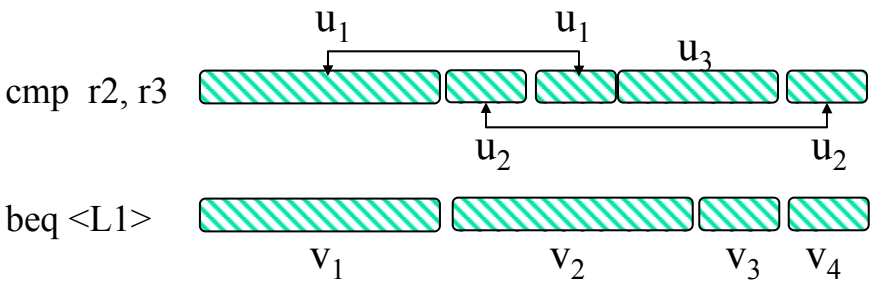


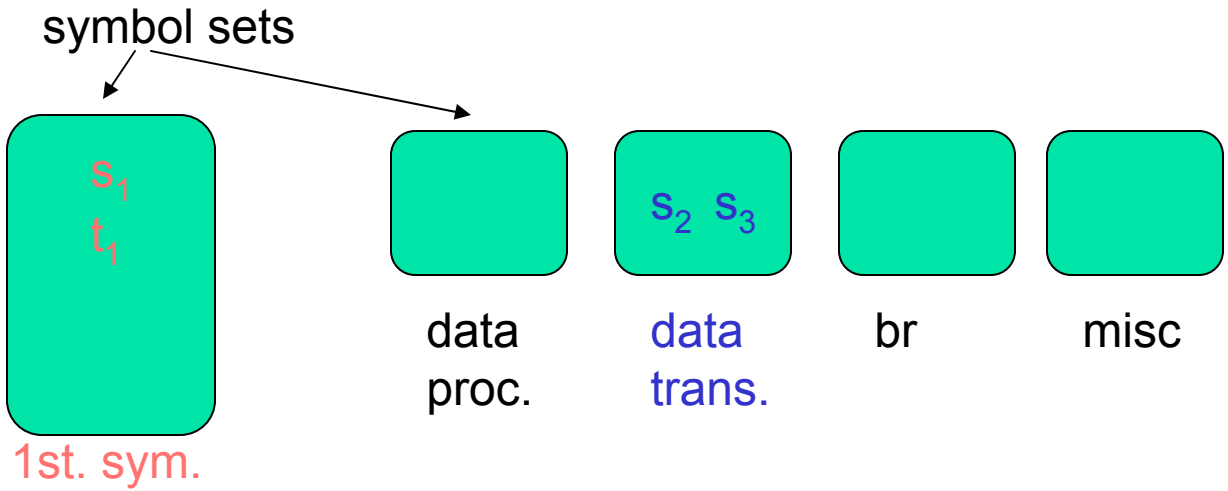


Modeling - Context Model (2)

itype model example:

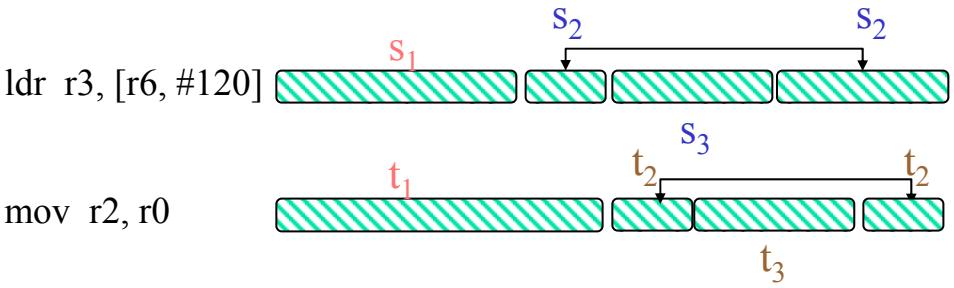


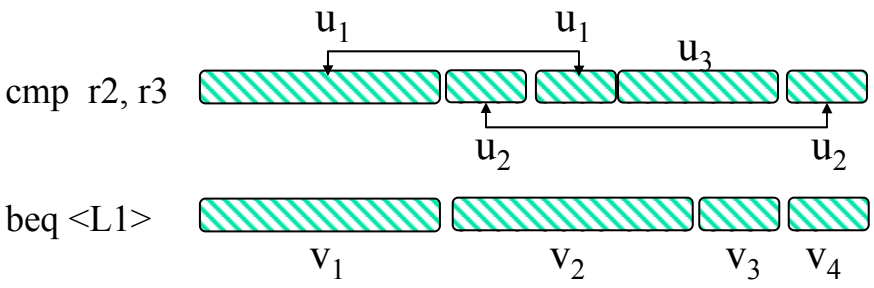


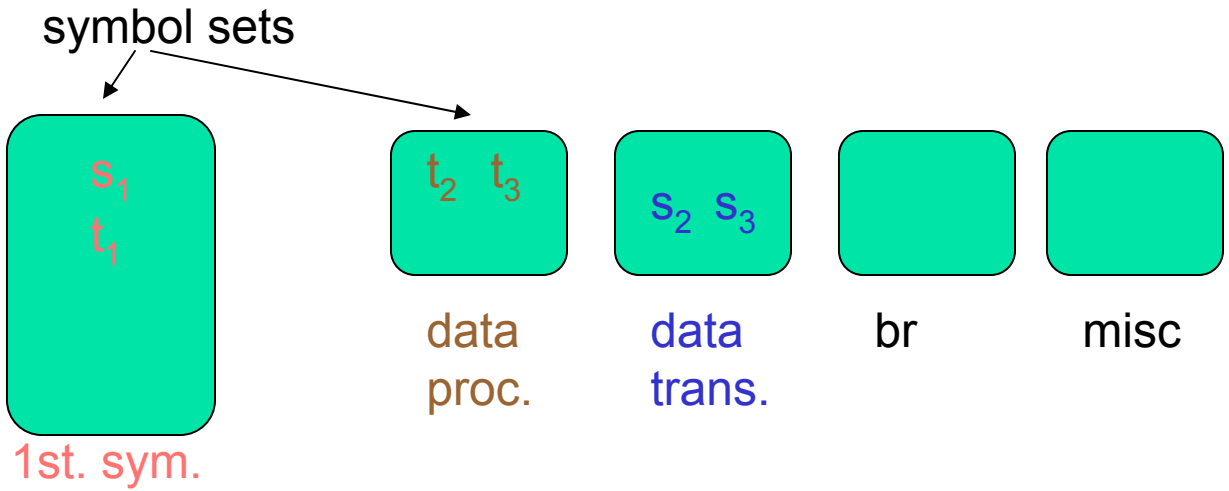


Modeling - Context Model (2)

itype model example:

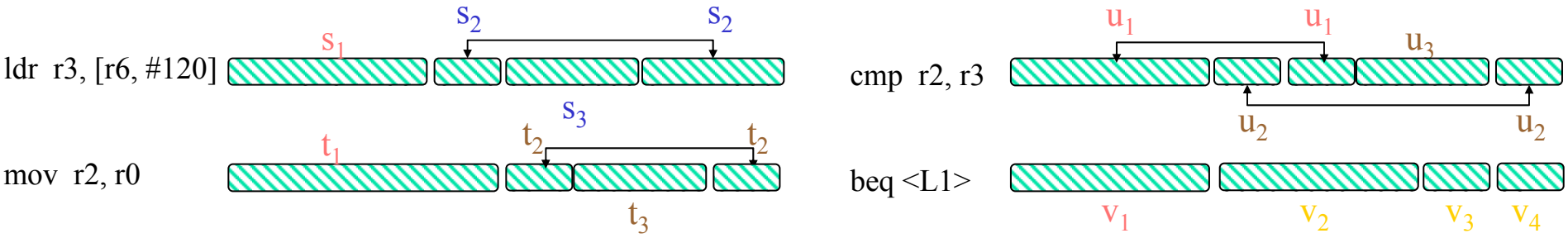


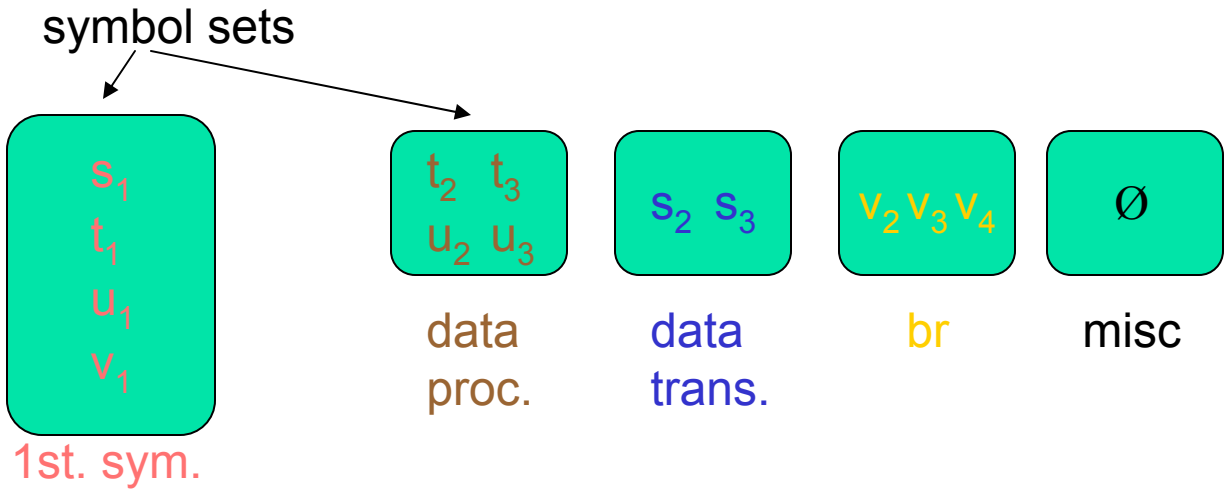




Modeling - Context Model (2)

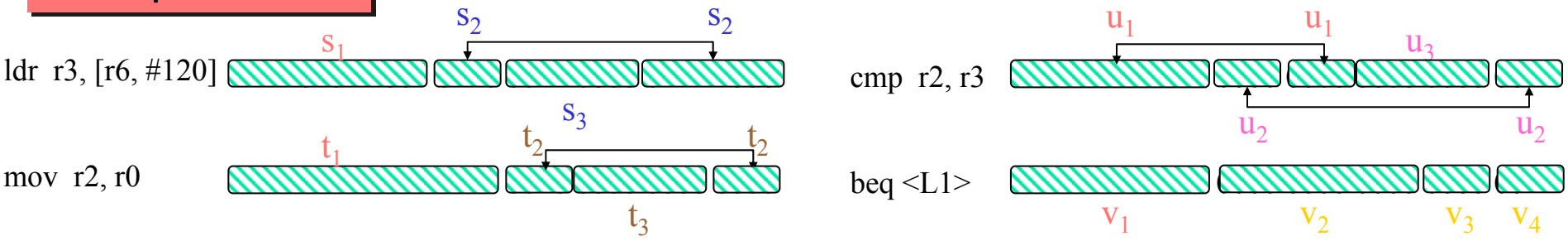
itype model example:



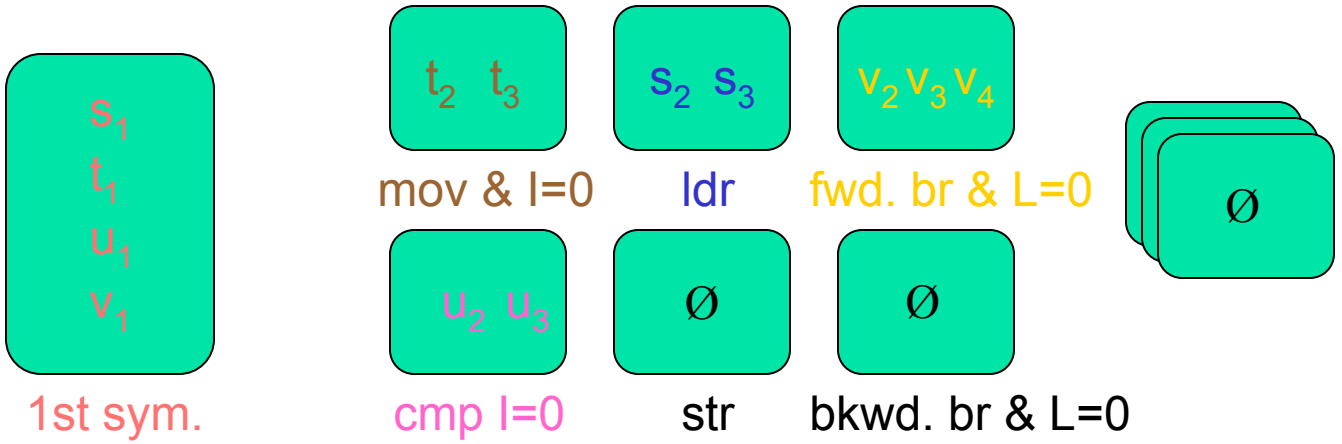


Modeling - Context Model (3)

imode model example:

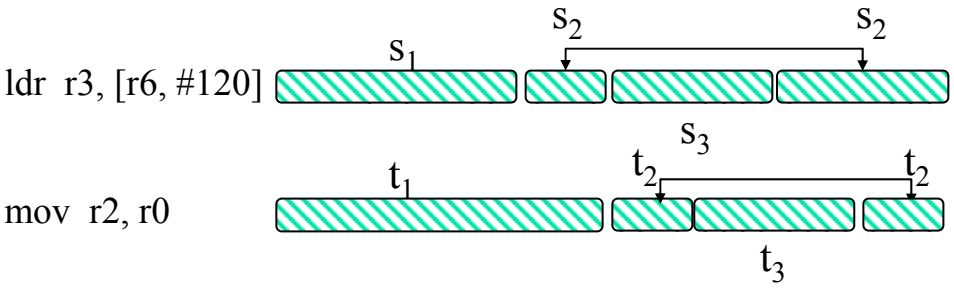


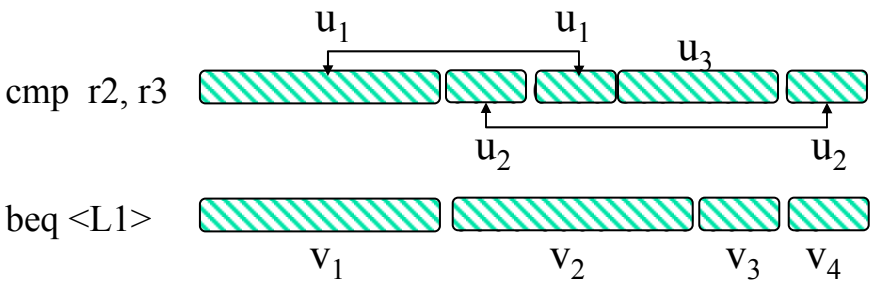
# of symbol sets: 14
(13 + 1)



Modeling - State Model

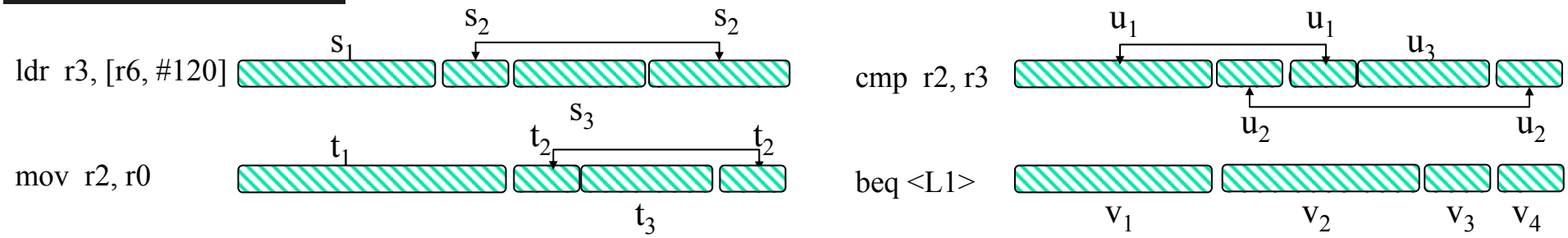
- Using *symbol position* to sort symbols

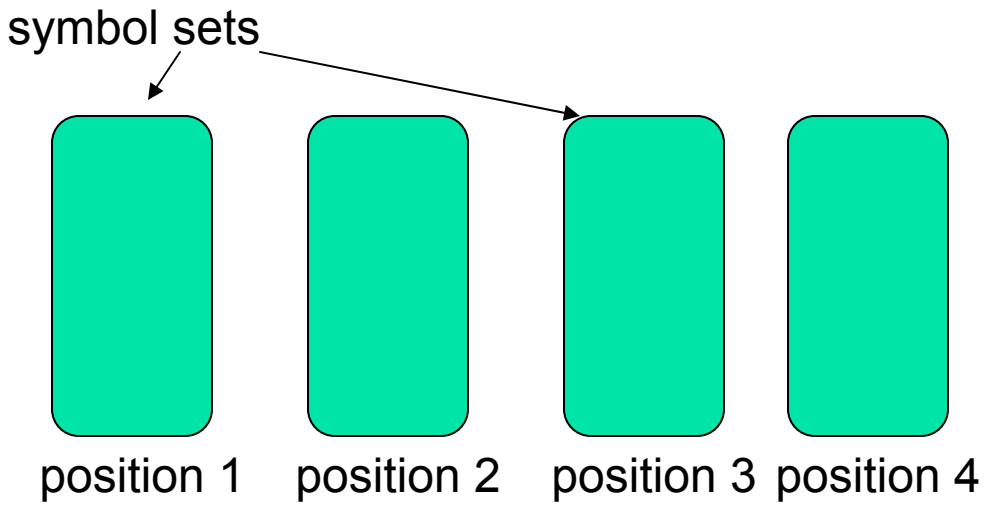




Modeling - State Model

state model example:

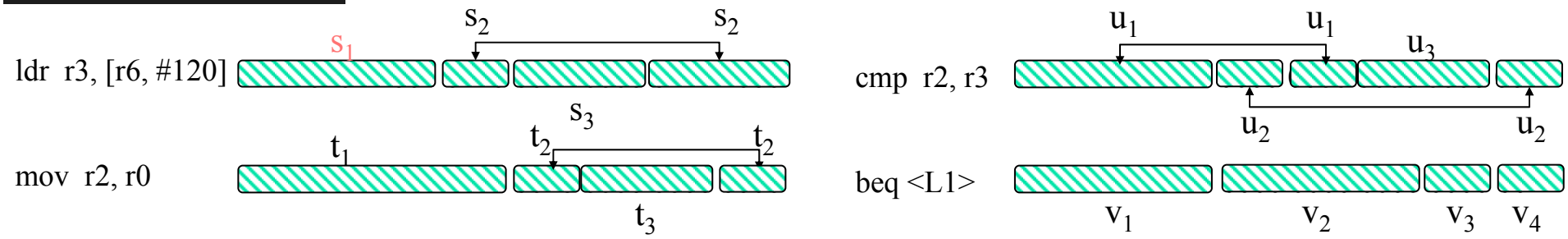


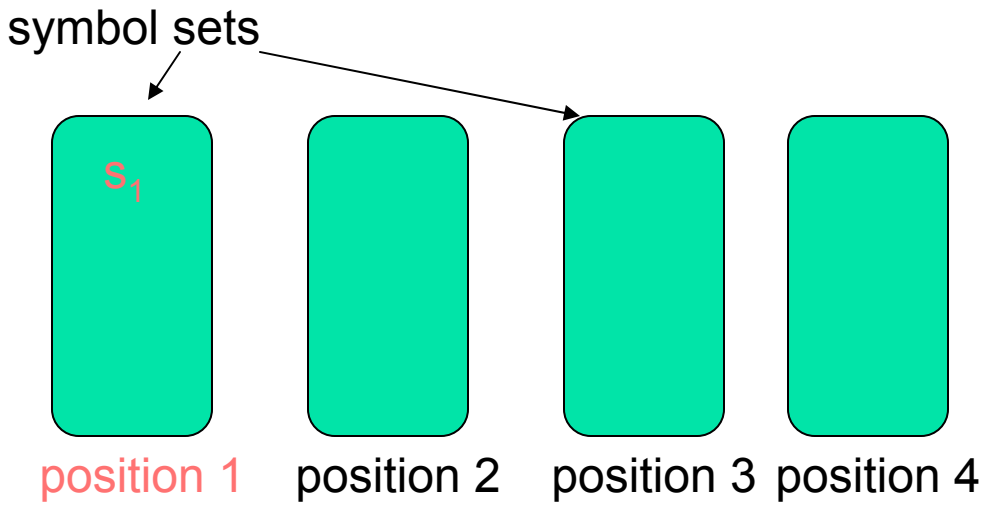


# of symbol sets: 4

Modeling - State Model

state model example:

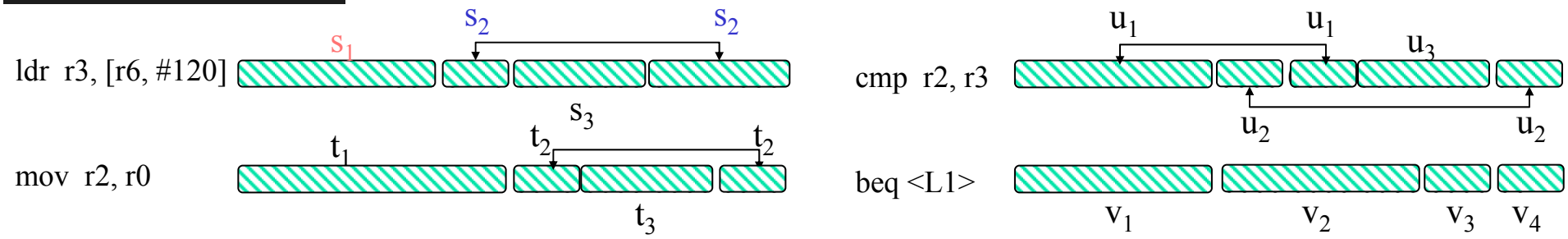


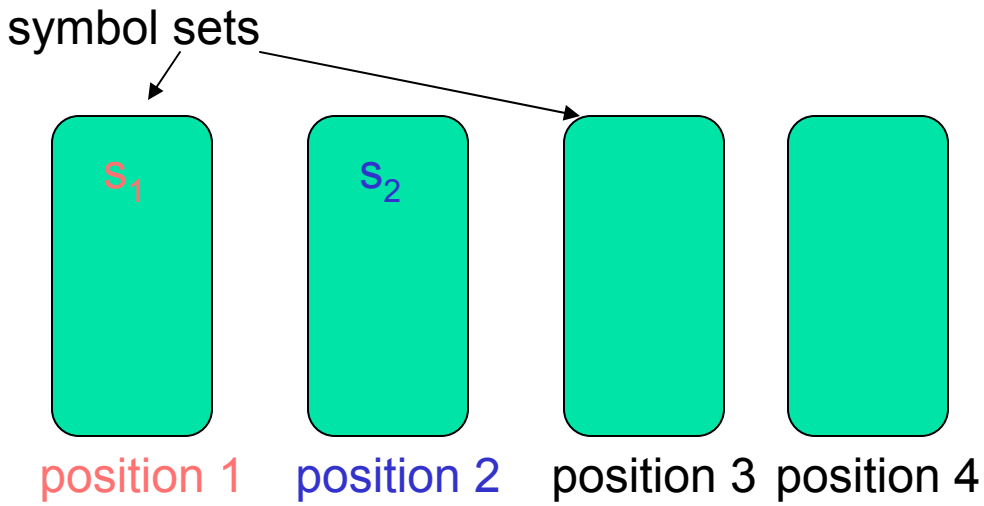


# of symbol sets: 4

Modeling - State Model

state model example:

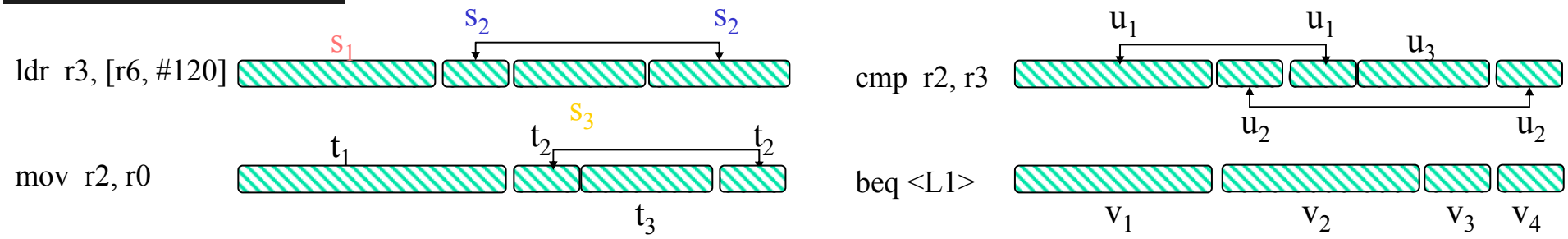


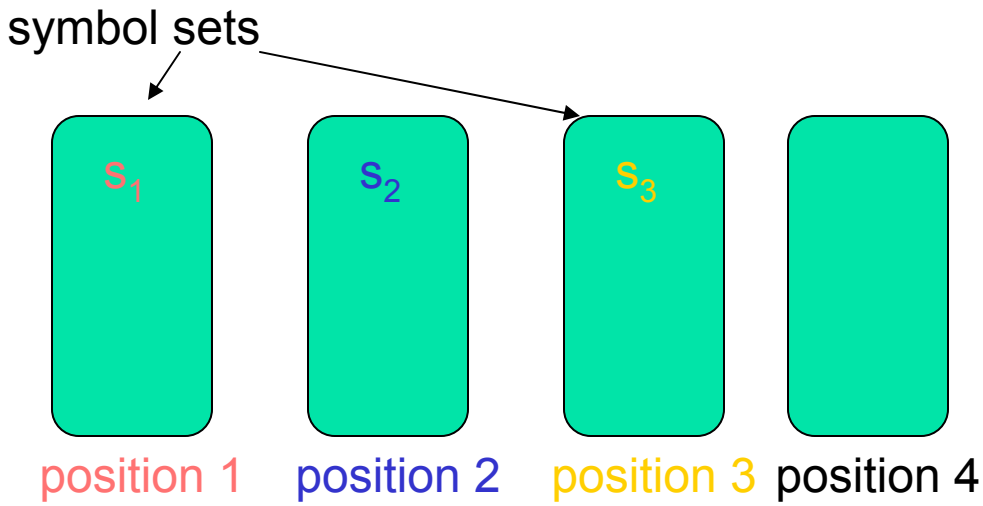


# of symbol sets: 4

Modeling - State Model

state model example:

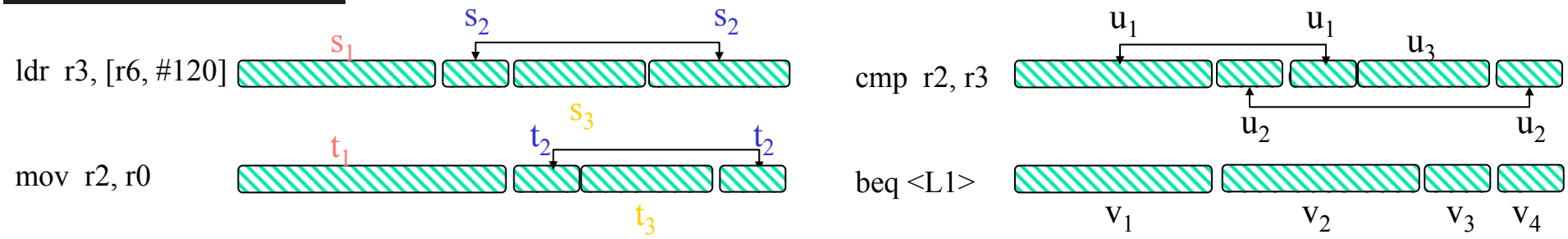


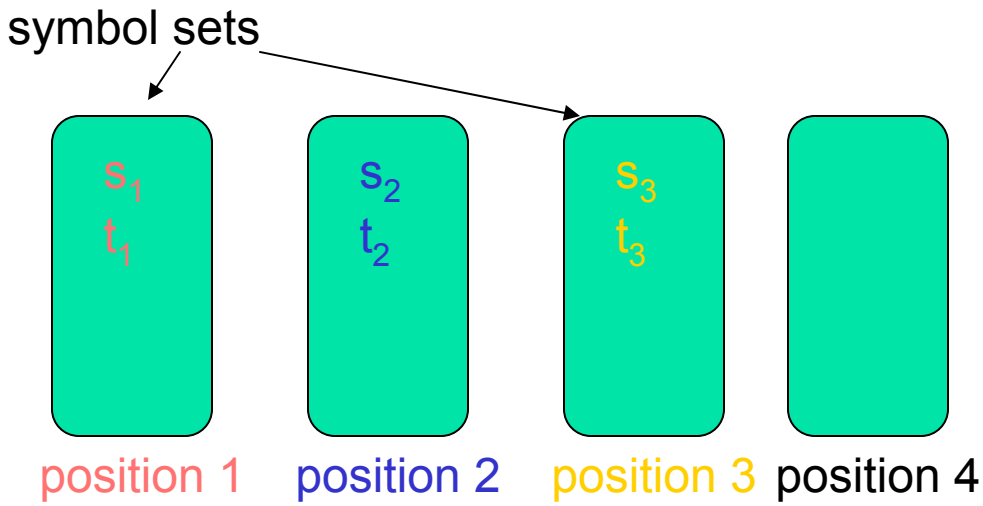


# of symbol sets: 4

Modeling - State Model

state model example:

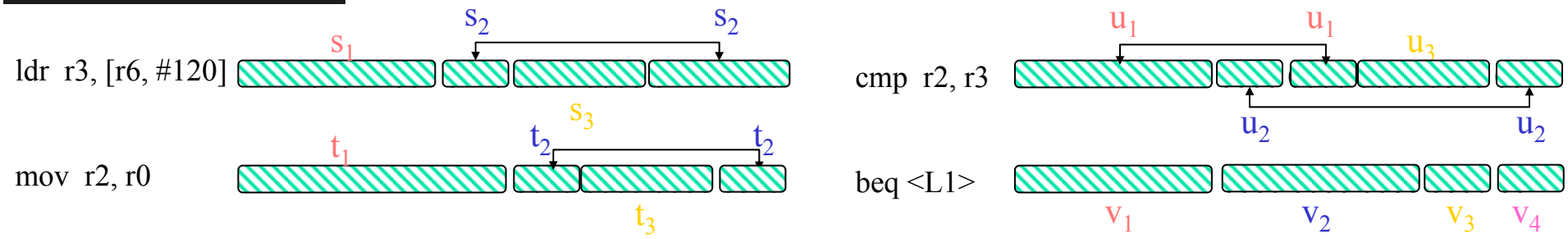


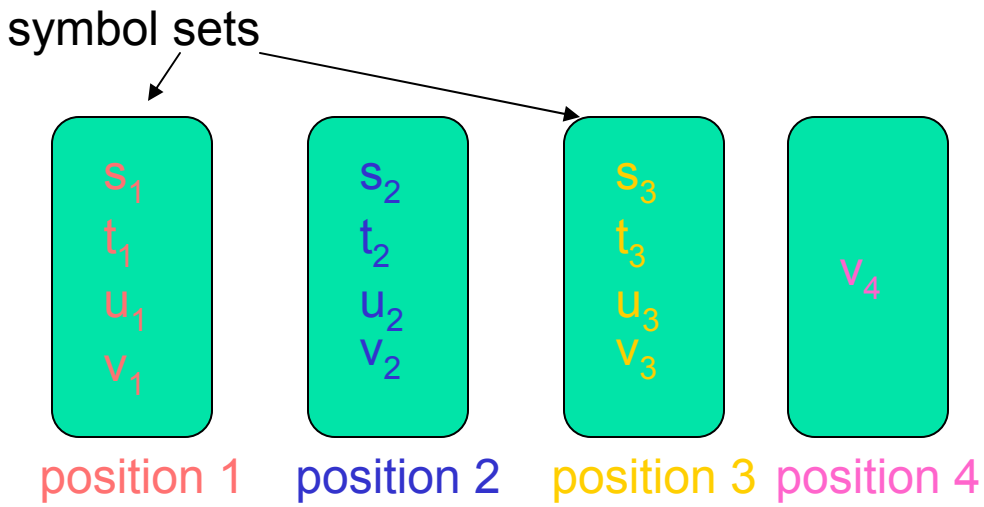


# of symbol sets: 4

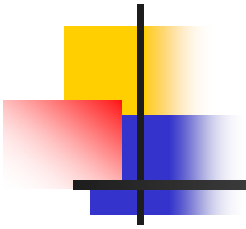
Modeling - State Model

state model example:





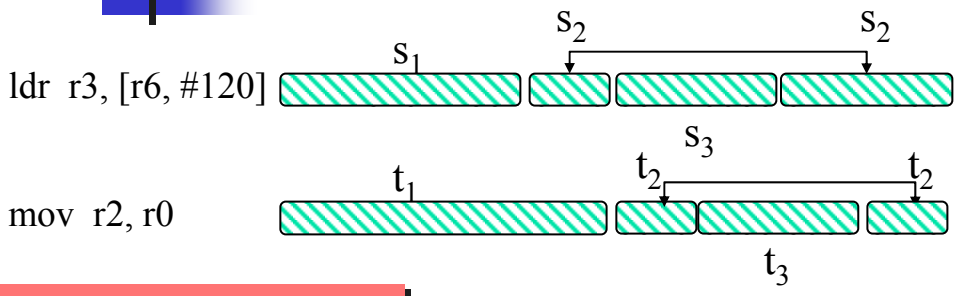
# of symbol sets: 4

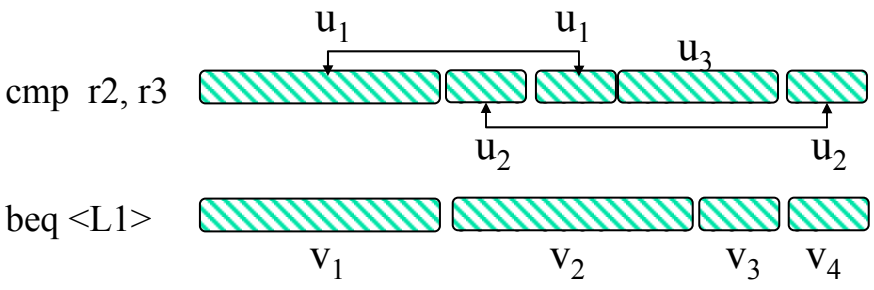


Modeling - Hybrid Model (1)

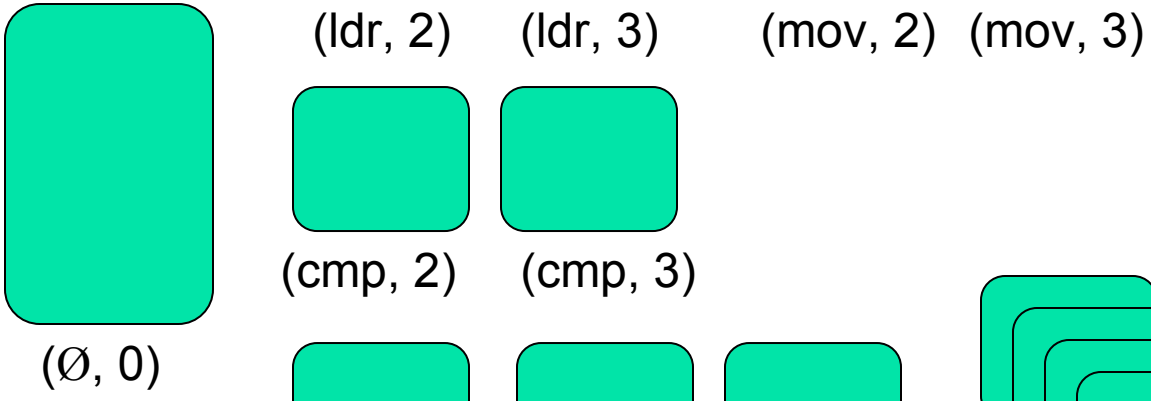
- Using both context (i.e. instruction class) and state (i.e. symbol position)
  - Each symbol is labeled with its *context* (inst. class) and its *state* (position).
  - The label defines a unique symbol set which the symbol is assigned to.

Modeling - Hybrid Model (2)





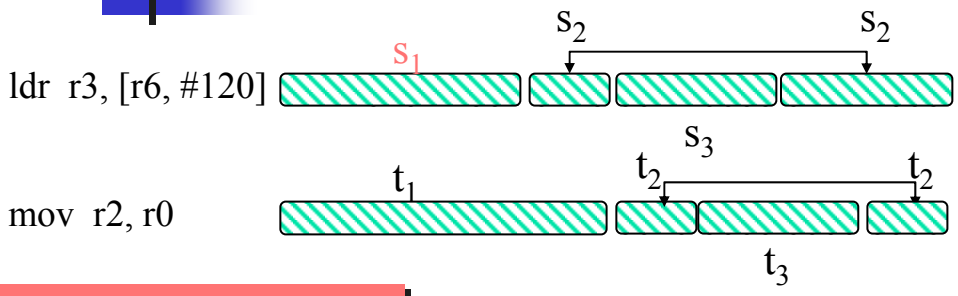
hybrid model example:

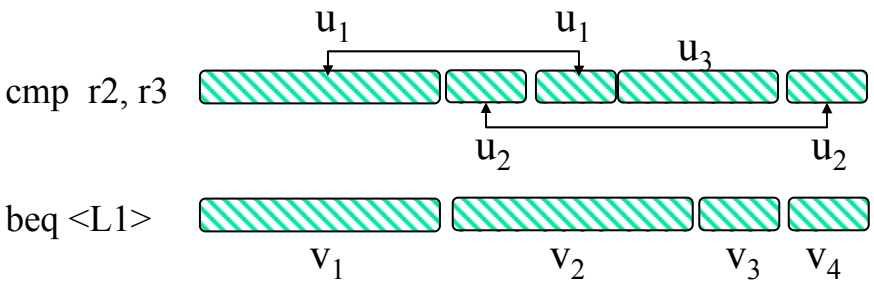


mov = mov, I=0 (M2)
cmp = cmp, I=0 (M1)
br = fwd. br, L=0 (M10)

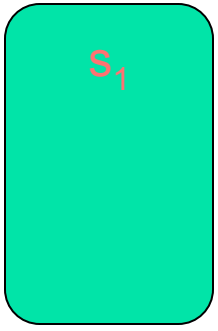
# of symbol sets: 33

Modeling - Hybrid Model (2)

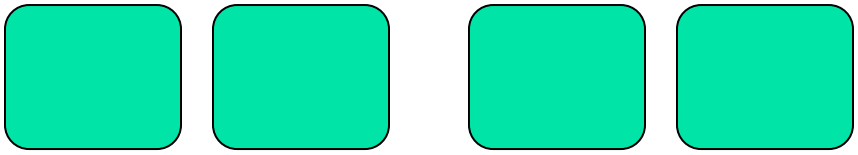




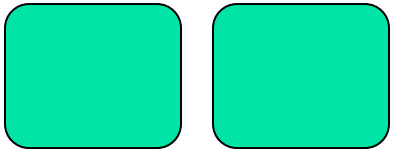
hybrid model example:



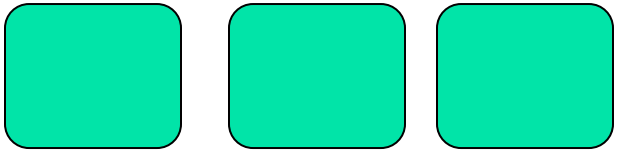
(Ø, 0)



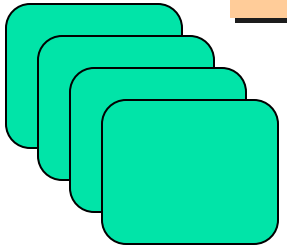
(ldr, 2) (ldr, 3) (mov, 2) (mov, 3)



(cmp, 2) (cmp, 3)



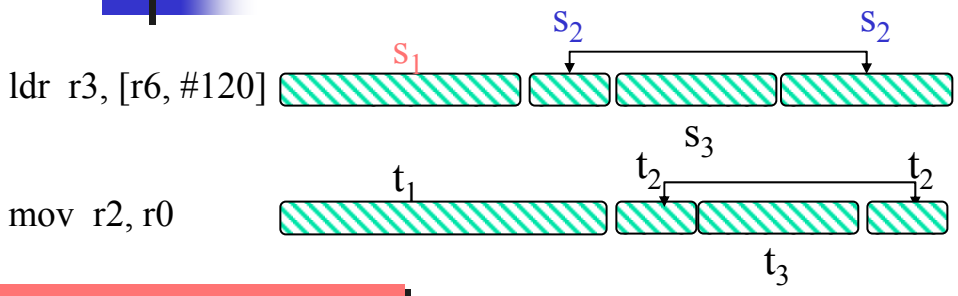
(br, 2) (br, 3) (br, 4)

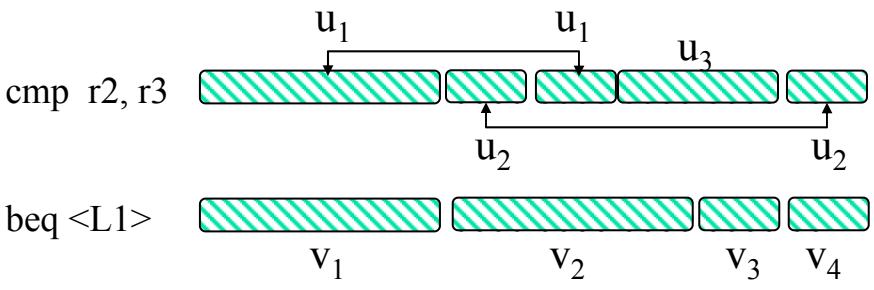


mov = mov, I=0 (M2)
cmp = cmp, I=0 (M1)
br = fwd. br, L=0 (M10)

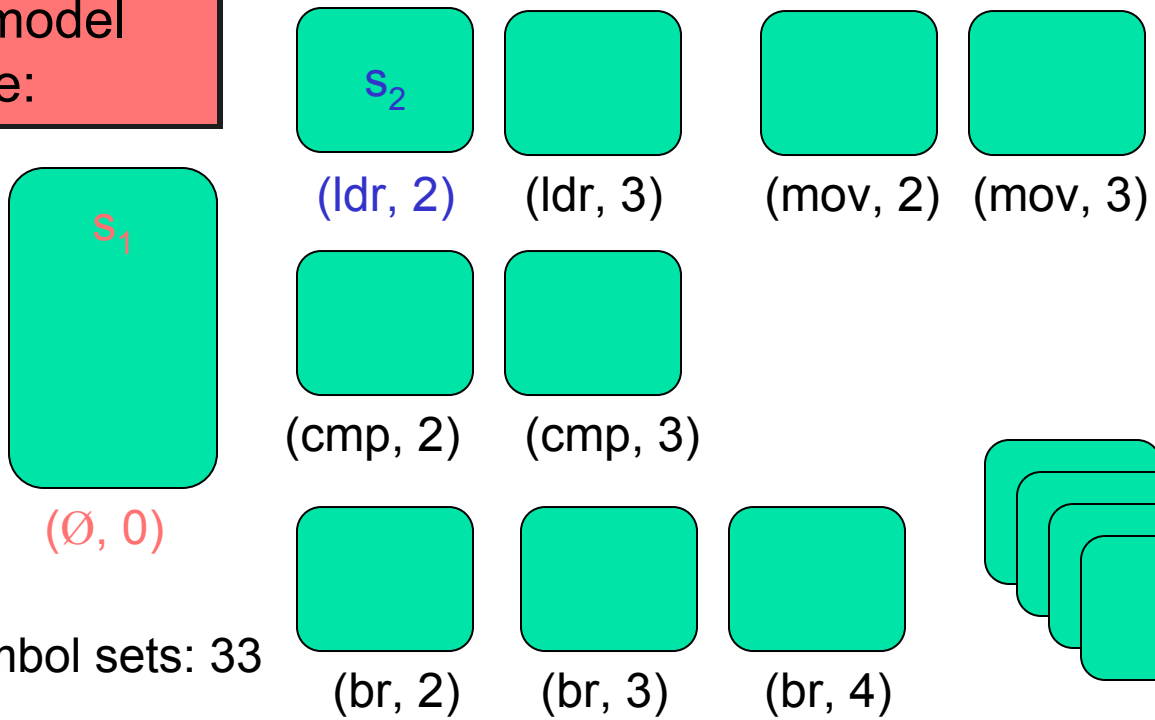
# of symbol sets: 33

Modeling - Hybrid Model (2)





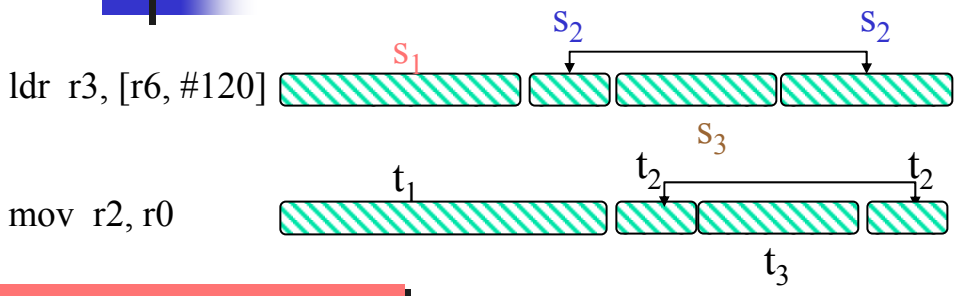
hybrid model example:

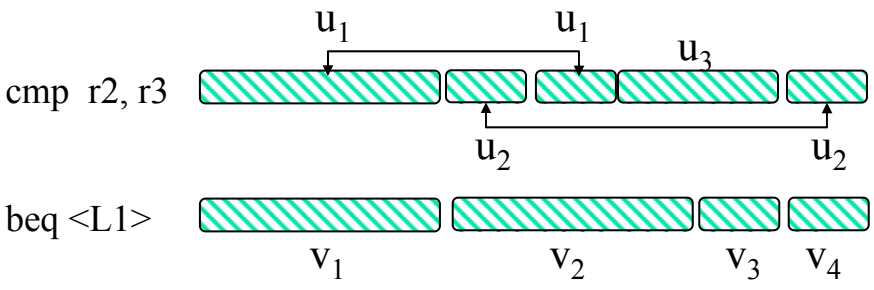


mov = mov, I=0 (M2)
cmp = cmp, I=0 (M1)
br = fwd. br, L=0 (M10)

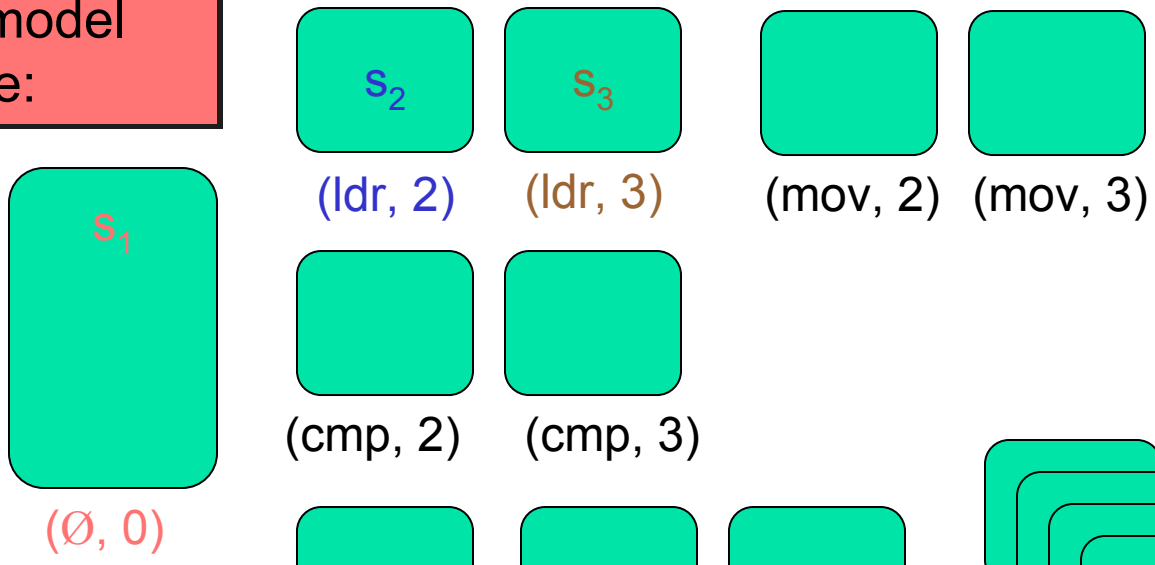
# of symbol sets: 33

Modeling - Hybrid Model (2)





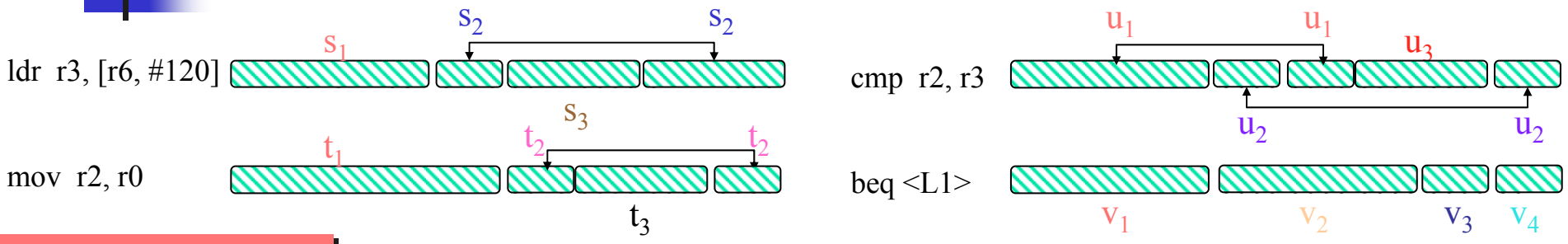
hybrid model example:



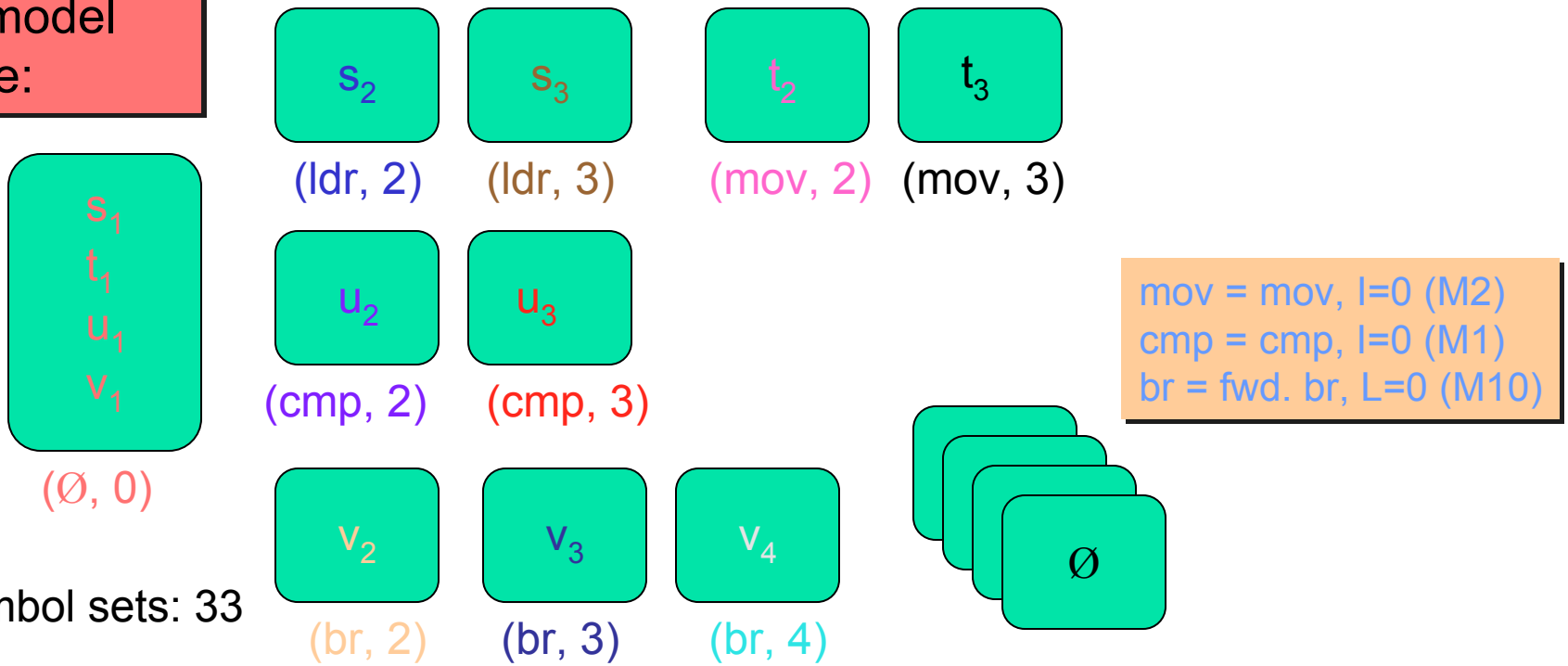
mov = mov, I=0 (M2)
cmp = cmp, I=0 (M1)
br = fwd. br, L=0 (M10)

# of symbol sets: 33

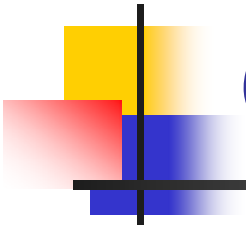
Modeling - Hybrid Model (2)



hybrid model example:

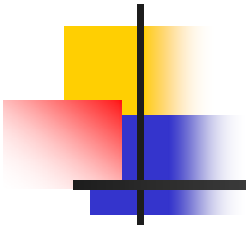


# of symbol sets: 33



Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

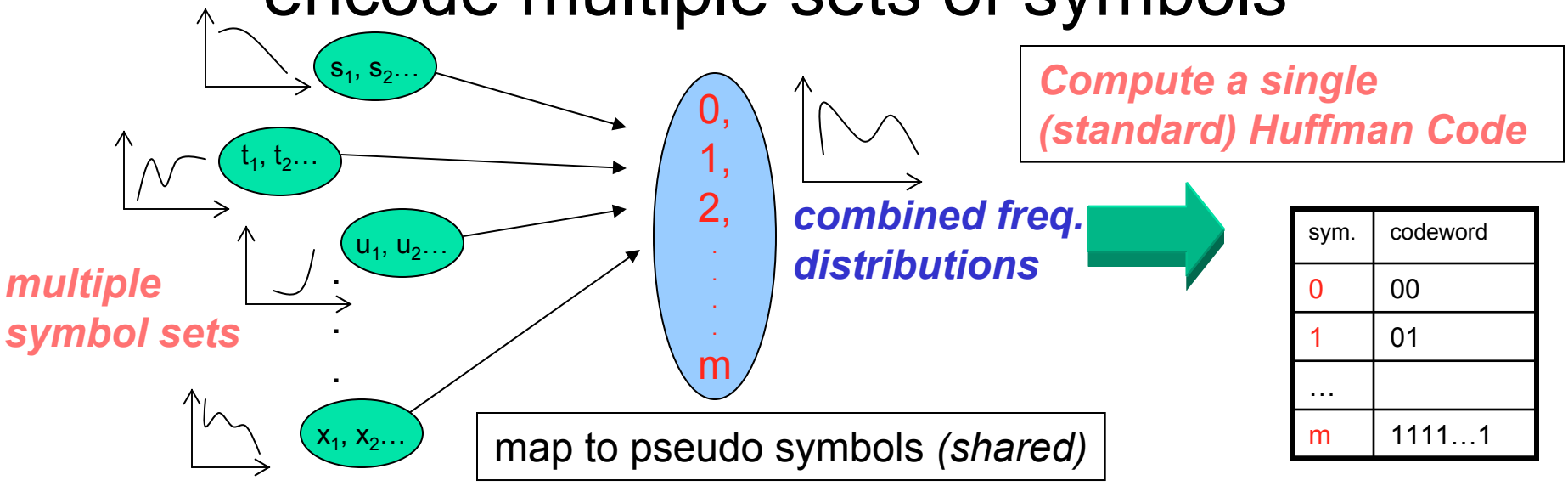


Encoding

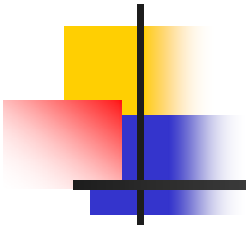
- Encode symbols based on their modeling results.
- Two options: *exact* or *curve-fitting* Huffman coding.
- Exact Huffman: for each set of symbols, a standard Huffman code is built.
  - \+ Optimal compression
  - \- Each symbol set requires a *distinct* Huffman code, and hardware decoder.

Encoding - Curve-fitting (1)

- Idea: use one Huffman code to encode all sets of symbols
- The code is the best single code to encode multiple sets of symbols

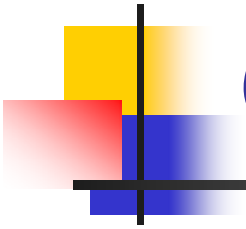


Encoding - Curve-fitting (2)



- Map symbols onto the pseudo symbols
  - Assigns the most frequent symbol to pseudo symbol 0, 2nd frequent sym. to 1, and so on.
- Combining frequency distributions
  - For a pseudo symbol i, its frequency is:

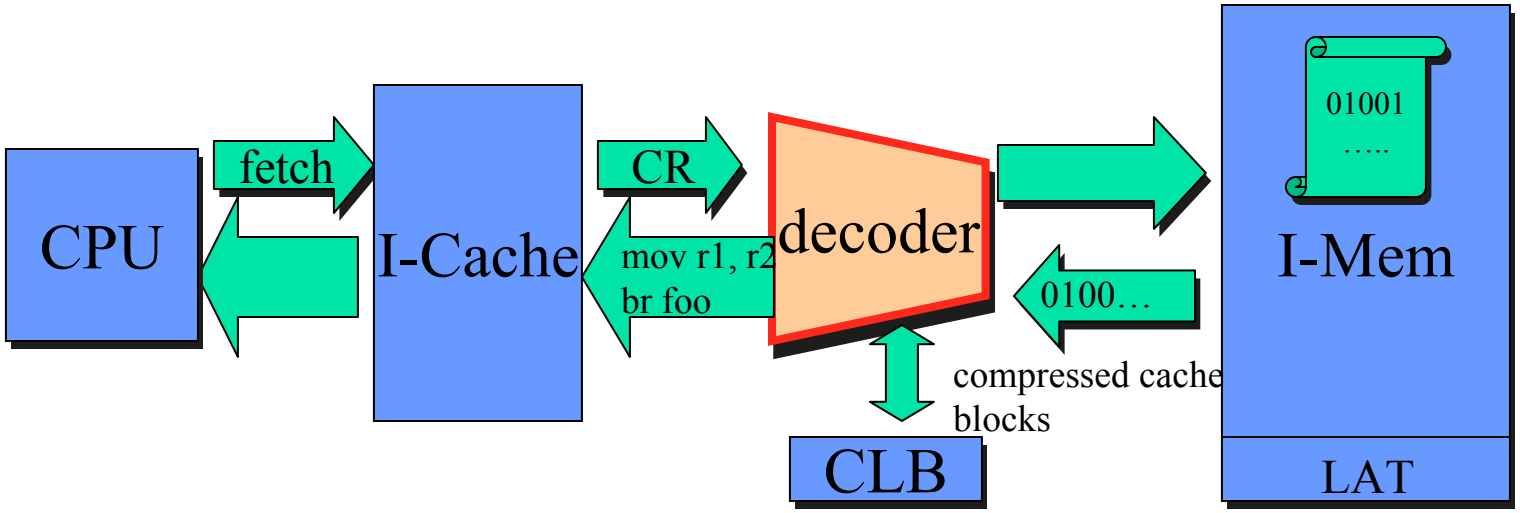
freq. of i = $\sum$ (freq. of (i+1)th frequent symbol in each set)



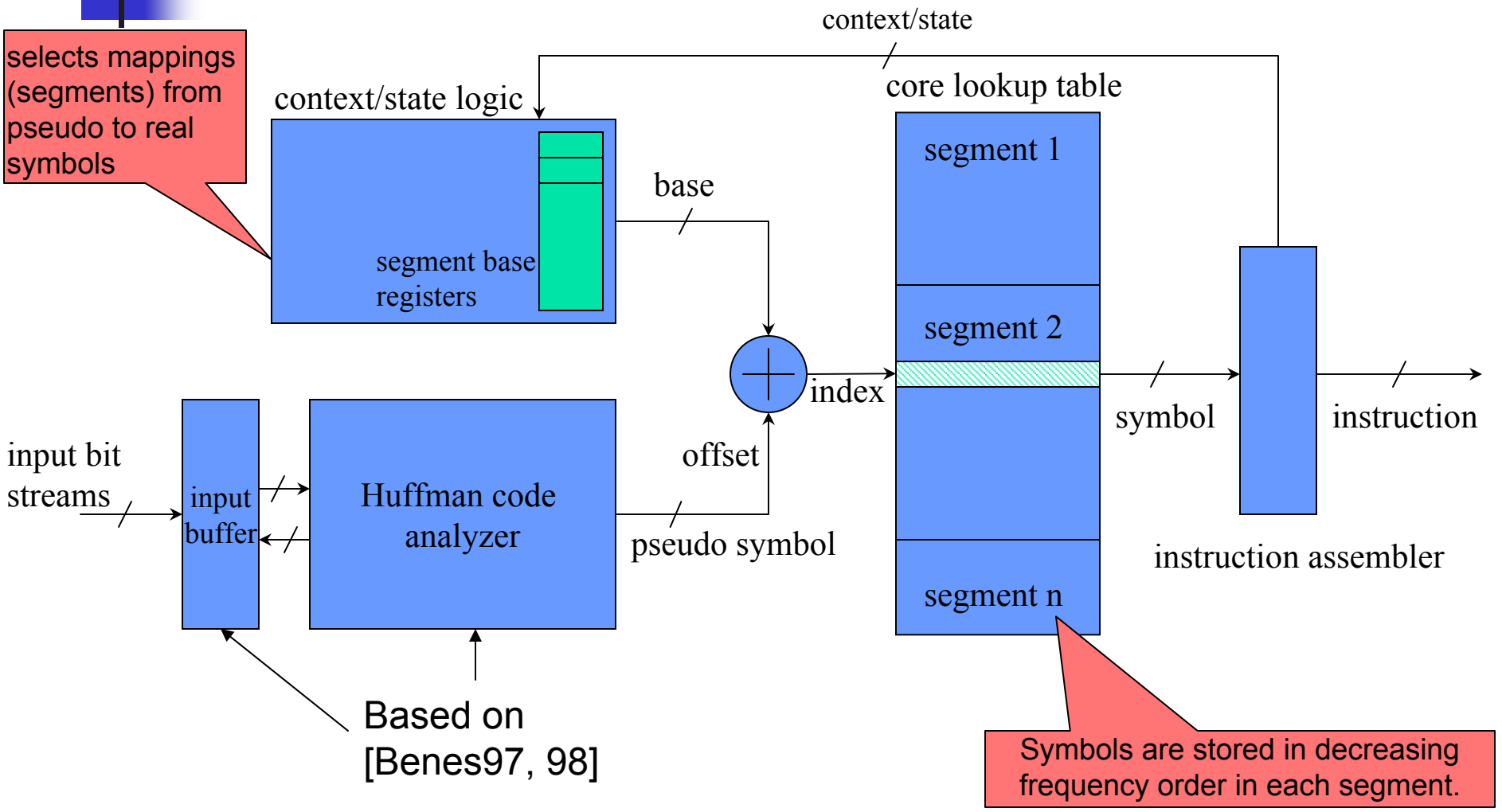
Outline

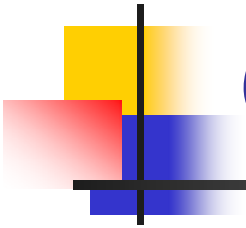
- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

Decoder Design (1)



Decoder Design (2)

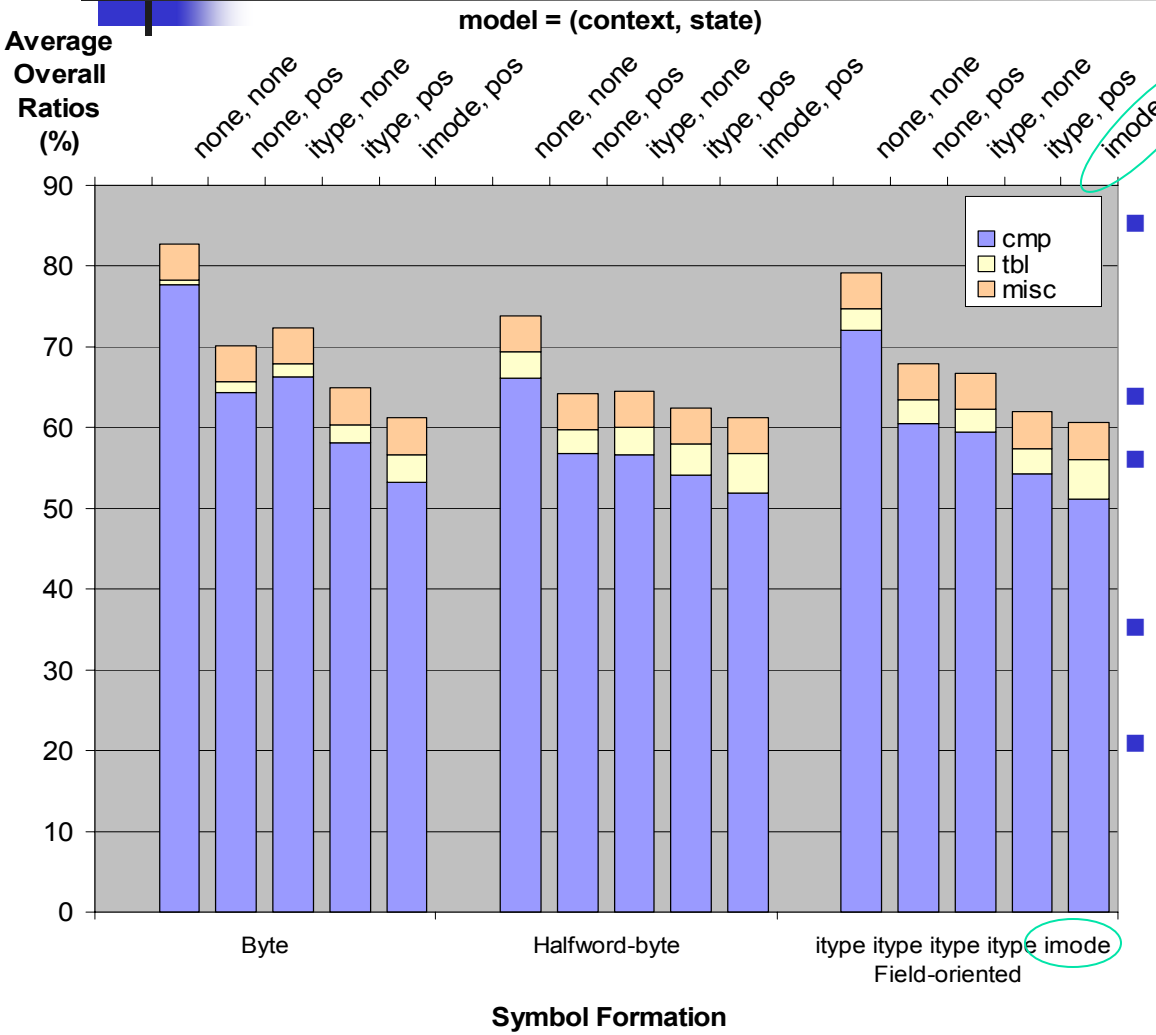




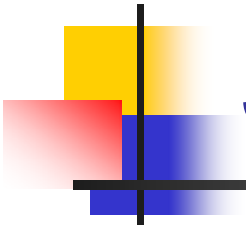
Outline

- Introduction
  - Motivation
  - Contributions
- Overview of Approach
- Compression Scheme
  - Symbol formation & instruction partitioning
  - Modeling
  - Encoding
- Decoder Design
- Results & Comparisons
- Conclusions and Future Work

Results

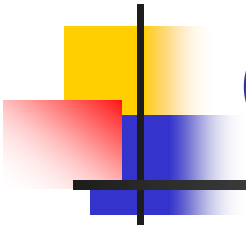


- **Setup:** applied to MediaBench, and other examples.
  - Trivial examples are excluded.
- Use *curve-fitting* code.
- **Best results:** *field-oriented* instruction partitioning (imode) and a *hybrid model*
- **Halfword-byte partitioning:** performs very well in all runs.
- **Field-oriented inst. partitioning:** only beats halfword-byte-byte in hybrid models.



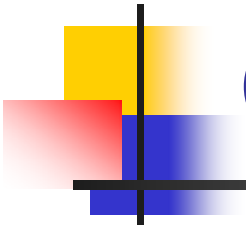
Summary of Our Results

- Compression result:
  - Single benchmark:
    - Best cmp. ratio: 48.1%
    - Best overall: 56.7% (incl. decoder hardware)
  - Average:
    - Overall: 59.4% (incl. decoder hardware)
- Curve-fitting performs well: cmp. ratio is only 2-3% more.
- No modification to processors/compilers
- Estimated impact on system performance (based on [Wolfe92]):
  - < 10% degradation
  - 2%-3% *improvement* in a best case scenario!



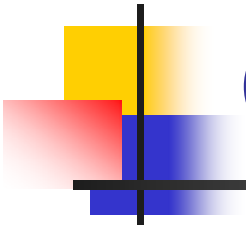
Comparisons (1)

- Leading results on MIPS: 33.8% [Lin03], 52%[Lekatsas00], 53.6%[Araujo00].
  - ARM is much harder to compress.
  - All require modifications to processor cores & compiler tools.
  - High decoder area overheads sometimes are not counted.



Comparisons (2)

- Leading results on ARM:
  - 46% [Lin02]
    - 10% overheads not counted (multiple Huffman codes)
    - Complex hardware (Huffman + dictionary decoders)
  - 56% [Lekatsas99]
    - complex arithmetic decoding, low decoding throughputs
    - less flexible to explore design space
  - Both approach require modifications to processor cores & compiler tools.
- *Our results are best reported for ARM which do not modify processor and compiler tools.*



Conclusions & Future Work

- A powerful code compression scheme.
- Simple and modular decoder design.
- The proposed symbol formation, modeling, and curve-fitting coding allows us to capture higher-order correlations with low decoding cost.
- Future:
  - *Systematic & automatic* symbol formation and instruction partitioning.
  - Models capturing *inter-instruction* correlations.