Build Your Own Static WCET Analyzer

the Case of the Automotive Processor AURIX TC275

2020-01-29 @ ERTS 2020

Wei-Tsun SUN*

ASTC-Design France; IRT Saint Exupéry, France; IRIT - University of Toulouse, France

Eric JENN

IRT Saint Exupéry, France; Thales AVS, France

Hugues Cassé

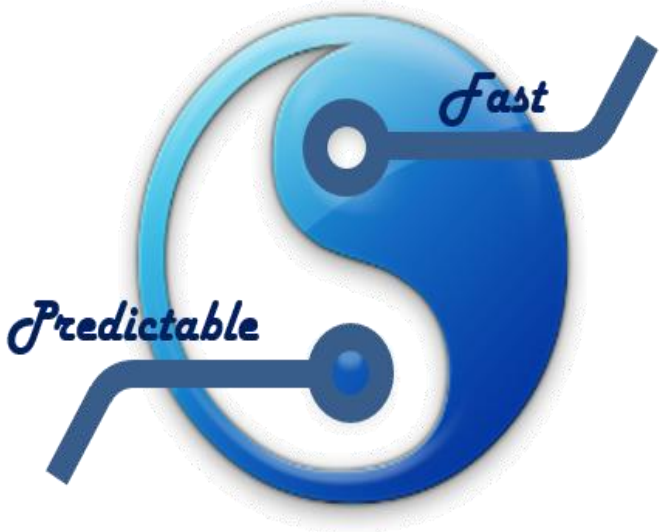
IRIT - University of Toulouse, France

- **This work is part of the project CAPHCA**
  - Supported by the ANR
  - safety (e.g. WCET)
  - performance (e.g. multicore-architecture)

Critical Applications
on Predictable High-Performance Computing Architectures

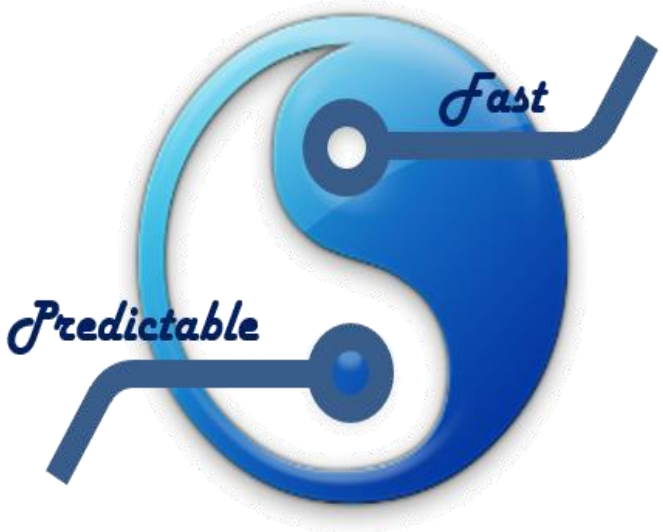
- **Papers in CAPHCA also presented in ERTS2**
  - **WE.2.C.3 @ 17h30** - Interferences in Time-Trigger Applications
  - **TH.1.C.2 @ 11h45** - Model checking for timing interferences
  - **FR.1.A.2 @ 09h30** - Design embedded SW in complex HW

- **This paper is under the collaboration and support of**
  - IRT Saint-Exupéry (stand 9), the **CAPHCA** team/project
  - IRIT - University of Toulouse, the team **TRACES**
  - ASTC-Design France (stand 38)



Critical Applications
on Predictable High-Performance Computing Architectures

Give a man a fish and you feed him for a day;
Teach a man to fish and you feed him for a lifetime



Critical Applications
on Predictable High-Performance Computing Architectures

Give a man a WCET and you save him for a day;
Teach a man to do WCET and you save him for a lifetime

- **Introduction – why need WCET (Worst-Case Execution Time)**
  - Scheduling within the system
  - Understanding the worst performance (or even power consumption)
- **Method of obtaining WCET**
  - Static – using processor models, abstract interpretation (math), e.g. OTAWA
  - Dynamic – measurements, statistics
- **Tools**
  - Industrial
  - academic (e.g. OTAWA – used in this talk)
- **This paper aims to reveal how one can implement and obtain WCET from a given architecture**
  - It covers a lot of aspects – from processor model, software structure….

- Look at OTAWA in 2 aspects
- User – uses OTAWA to get WCET
- Developer – add features to OTAWA to support WCET estimation for a given architecture

- Few aspects of OTAWA
  - From the TRACES team, IRIT, University of Toulouse
  - Main tech: Abstract interpretation
    - Consider all possible state (scenarios)
    - In an abstract manner
    - Not rely on input sequences
  - HW: Need to model processor
    - Pipelines
    - Memory hierarchy
    - Component which enhance the performance
  - SW: the information of the program
  - Backup plan:
    - Assume for the worst
    - Over-estimation

- **In CAPHCA we want to get WCET from Infineon TC275**

- **The problem**
  - Obtain safe WCET estimations for TC275 (a multi-core platform)

- **The solution**
  - Add support for TC275 to OTAWA
  - Identify the characteristics/behaviours of hardware
    - Fetch-FIFO: to decrease the penalty of program cache-misses
    - Write-buffer: to decouple the CPU ops and memory access
    - Both are not well-documented in the user-manual
  - Provide means to increase the precision of the results
    - Reduce over-estimation, keep safety (i.e. estimation never < actual)
  - Provide guidelines to “adapt” OTAWA for a new architecture

❖ Computing WCET with OTAWA

- ❖ OTAWA takes the program binary as the input
  - ❖From our academic partner: TRACES team from IRIT

- ❖Provides WCET (in CPU cycles)
- ❖However, a binary does not have information about hardware
  - ❖ same binary, different hardware -> different speed.
  - ❖ process the same instruction in different number of cycles

Synopsis

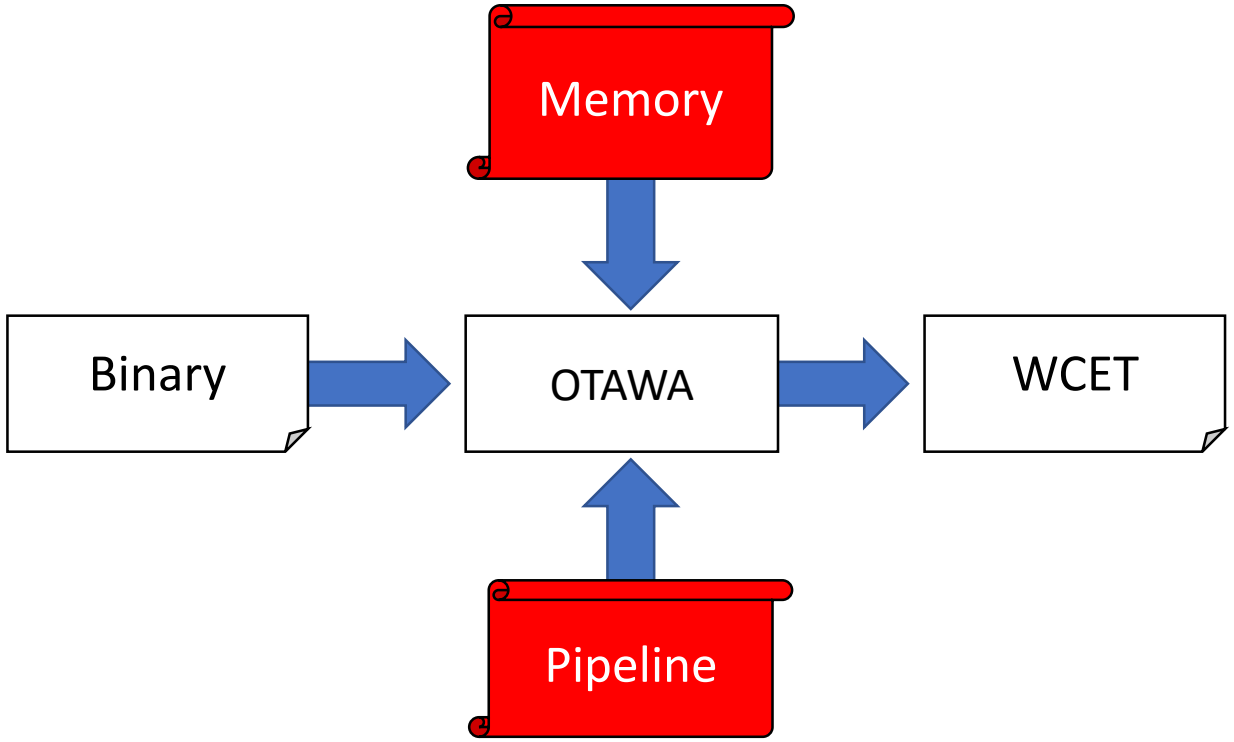
```
__m128d _mm_mul_pd (__m128d a, __m128d b)
#include <emmintrin.h>
Instruction: mulpd xmm, xmm
CPUID Flags: SSE2
```

Performance

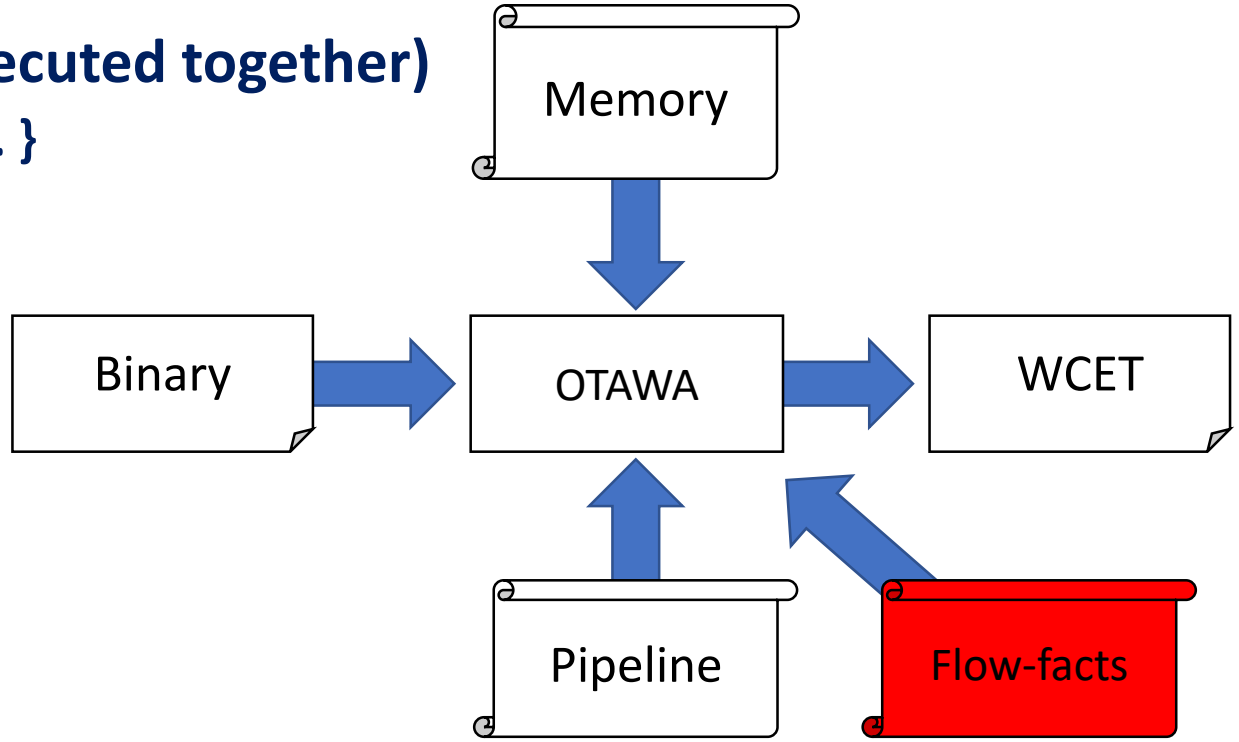
Architecture	Latency	Throughput (CPI)
Skylake	3	0.5
Broadwell	5	0.5
Haswell	5	0.5
Ivy Bridge	5	1

❖ Need to provide hardware information

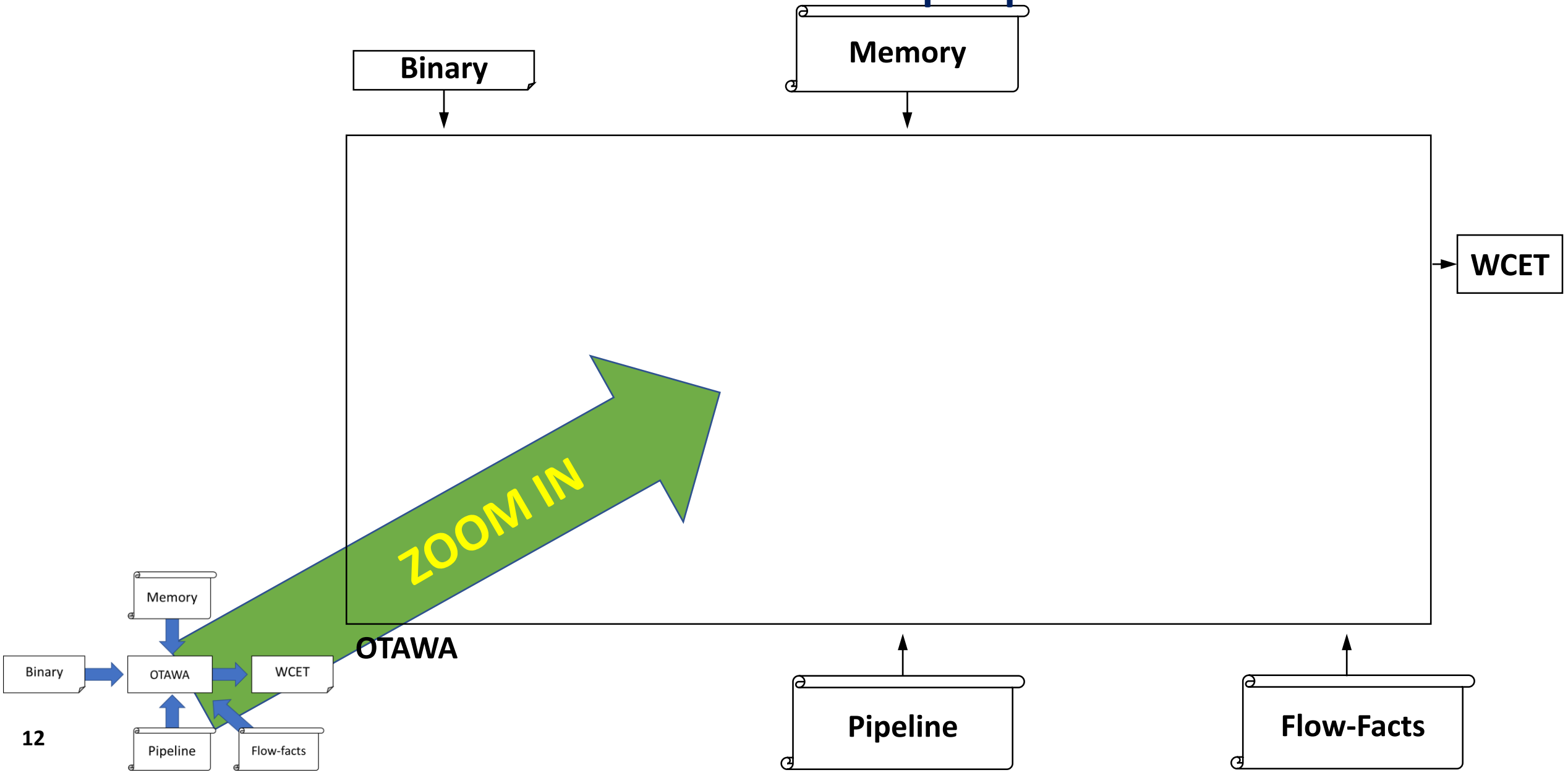
- ❖ Pipeline
  - ❖ How many stages, latencies, types
- ❖ Memory
  - ❖ Type of the memory – cache, scratch-pad, …
  - ❖ Latency - (remember we are doing WCET…)
  - ❖ Size – important especially for cache



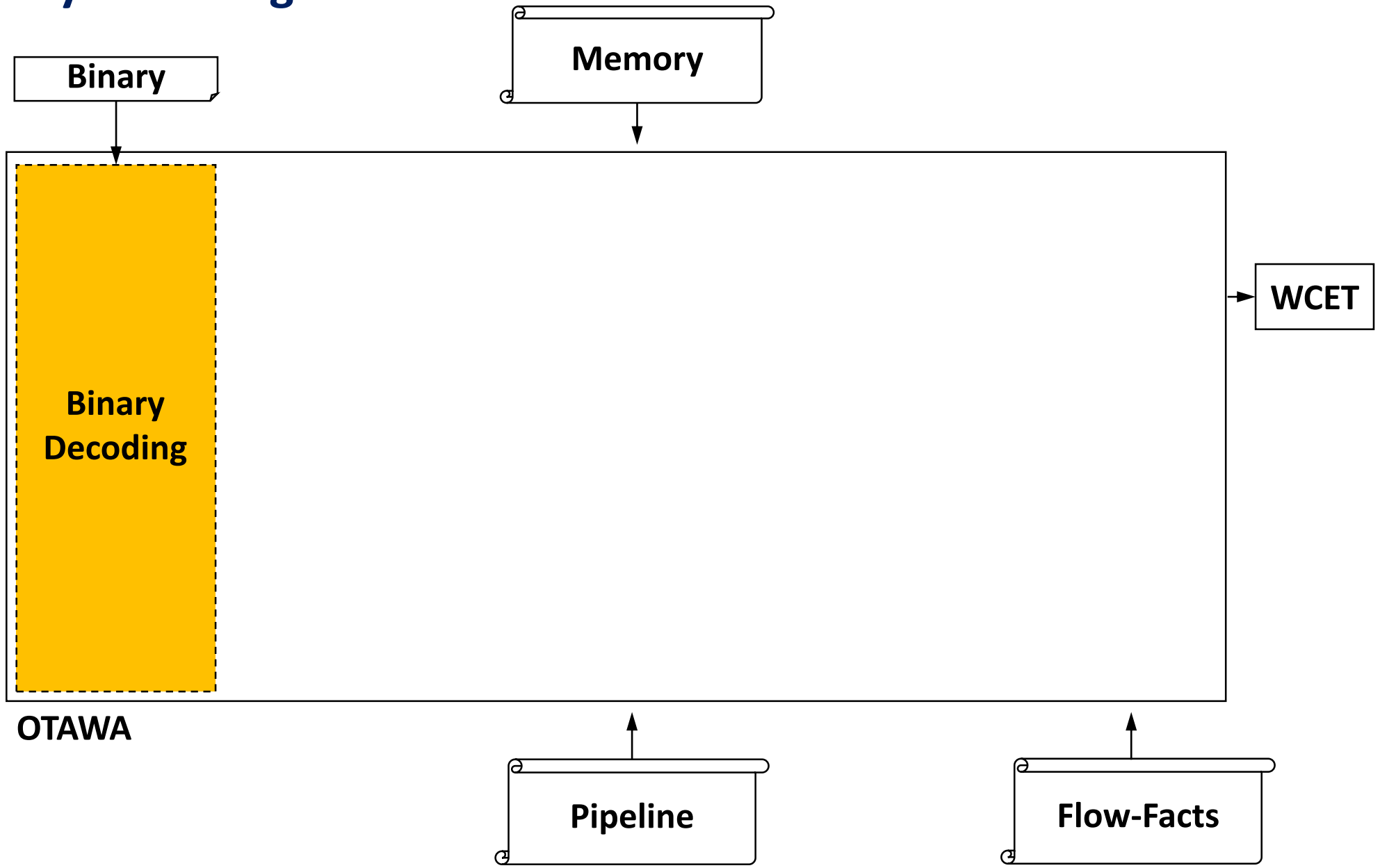
- Even though we have the binary
  - Still don’t know certain aspects (available at runtime)
  - Loop iterations – maximum times a loop will do
  - Branch targets: e.g. switch-cases, function pointers
  - Infeasible paths (some path never be executed together)
    - if (a != 0) { … } … some codes; if (a == 0) { …. }
  - Provide these information as flow-facts



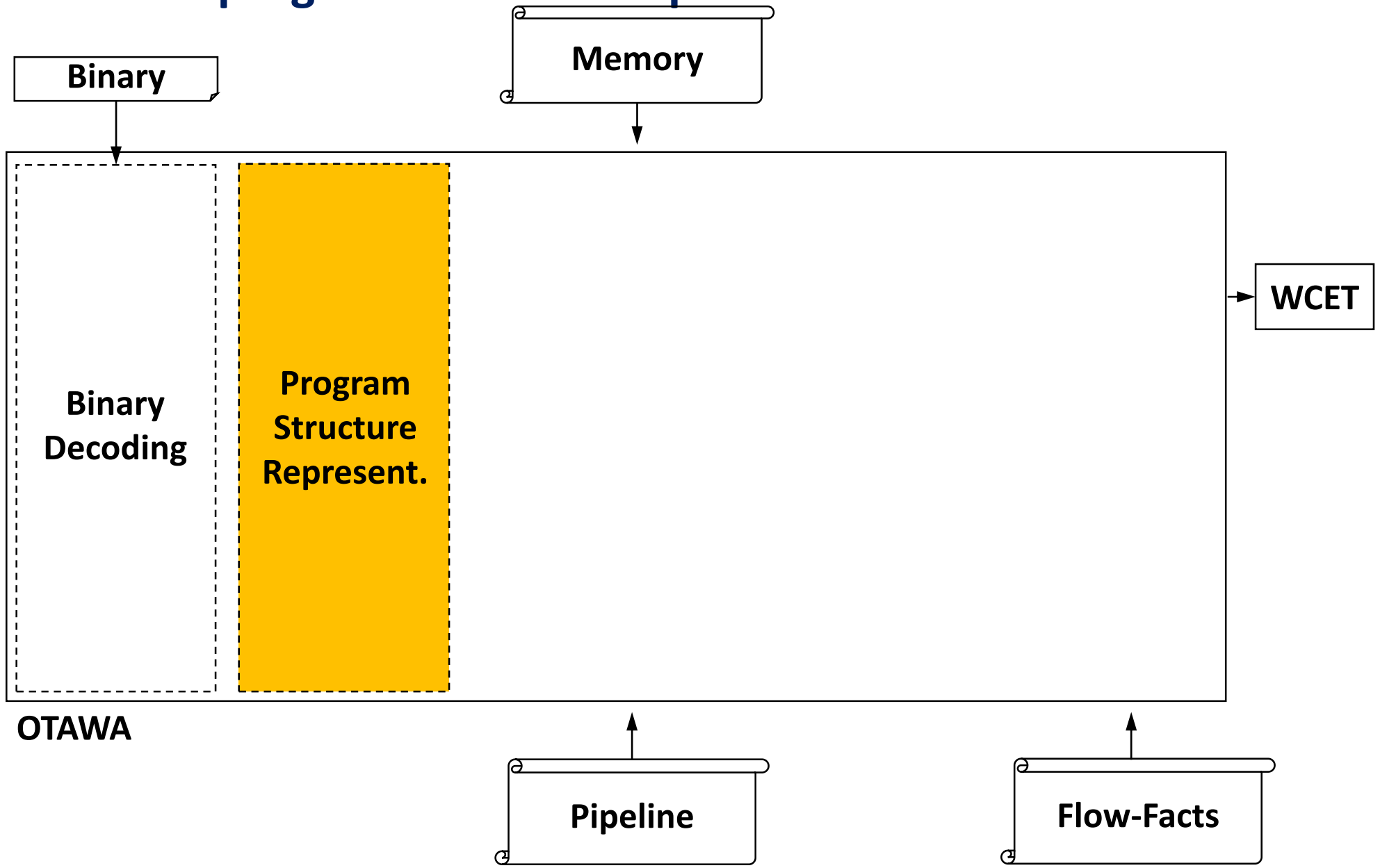
❖ More details of WCET estimation – a user's perspective.



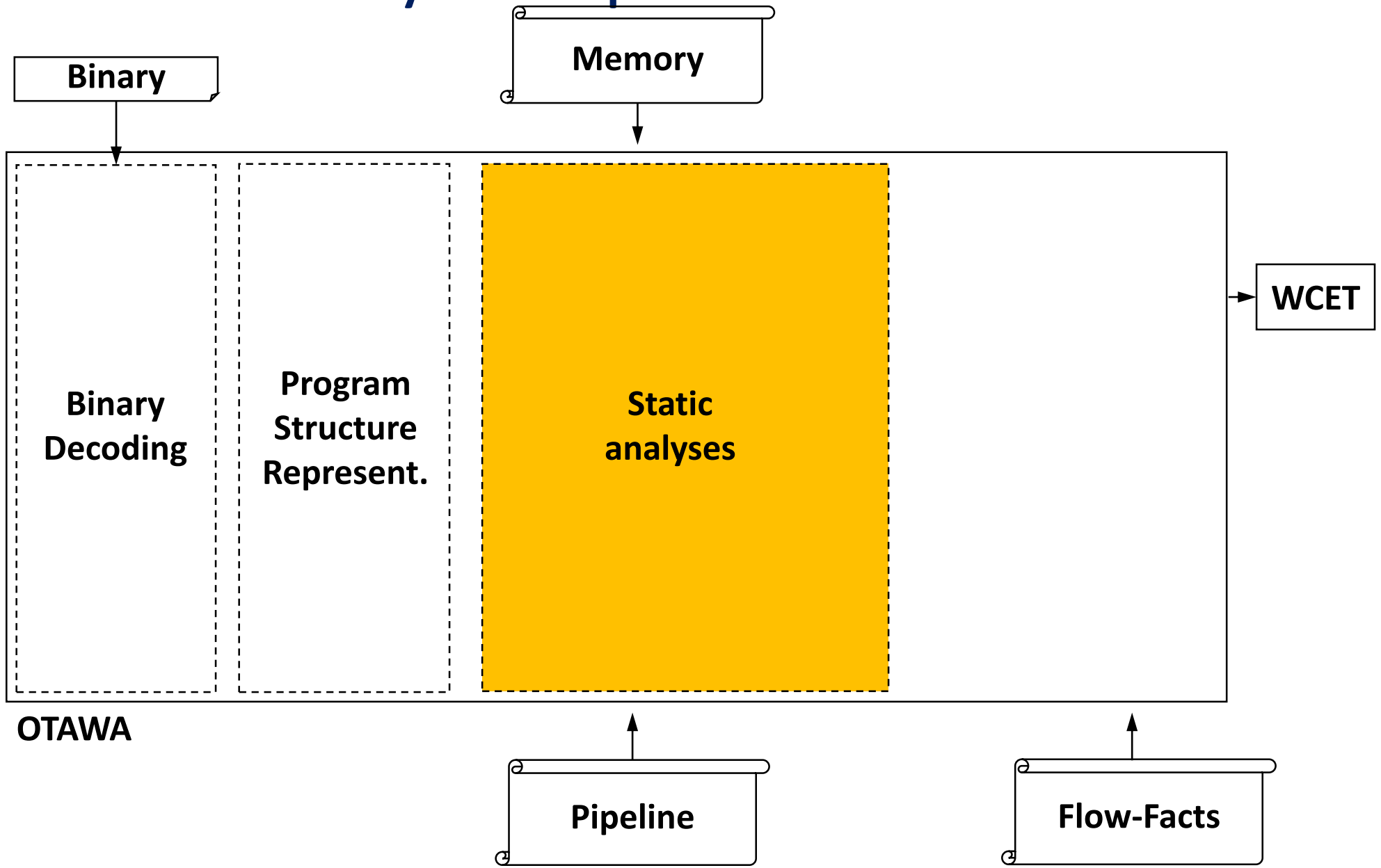
❖ First step: Binary Decoding



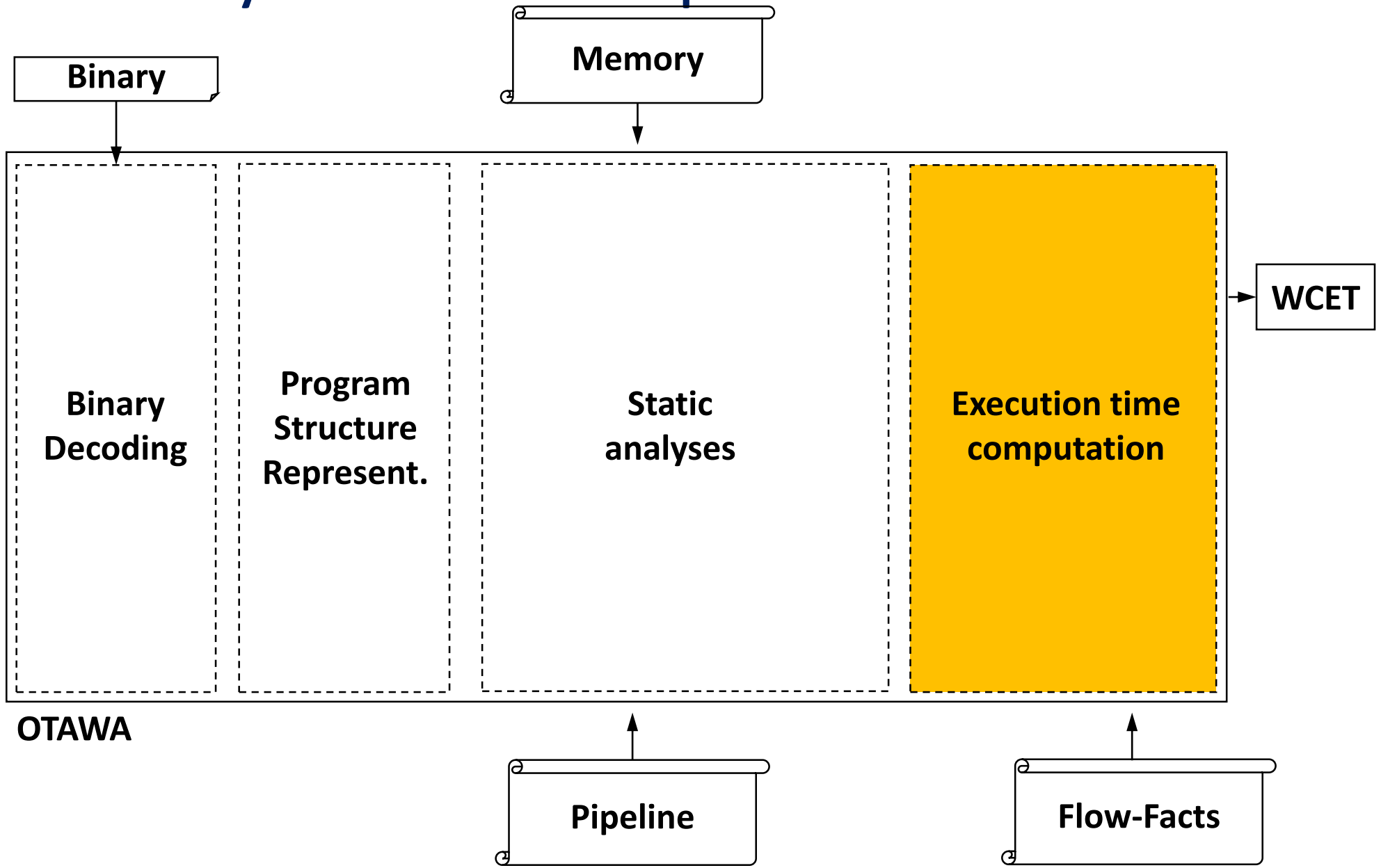
❖ 2nd step: represent the program so it can be processed



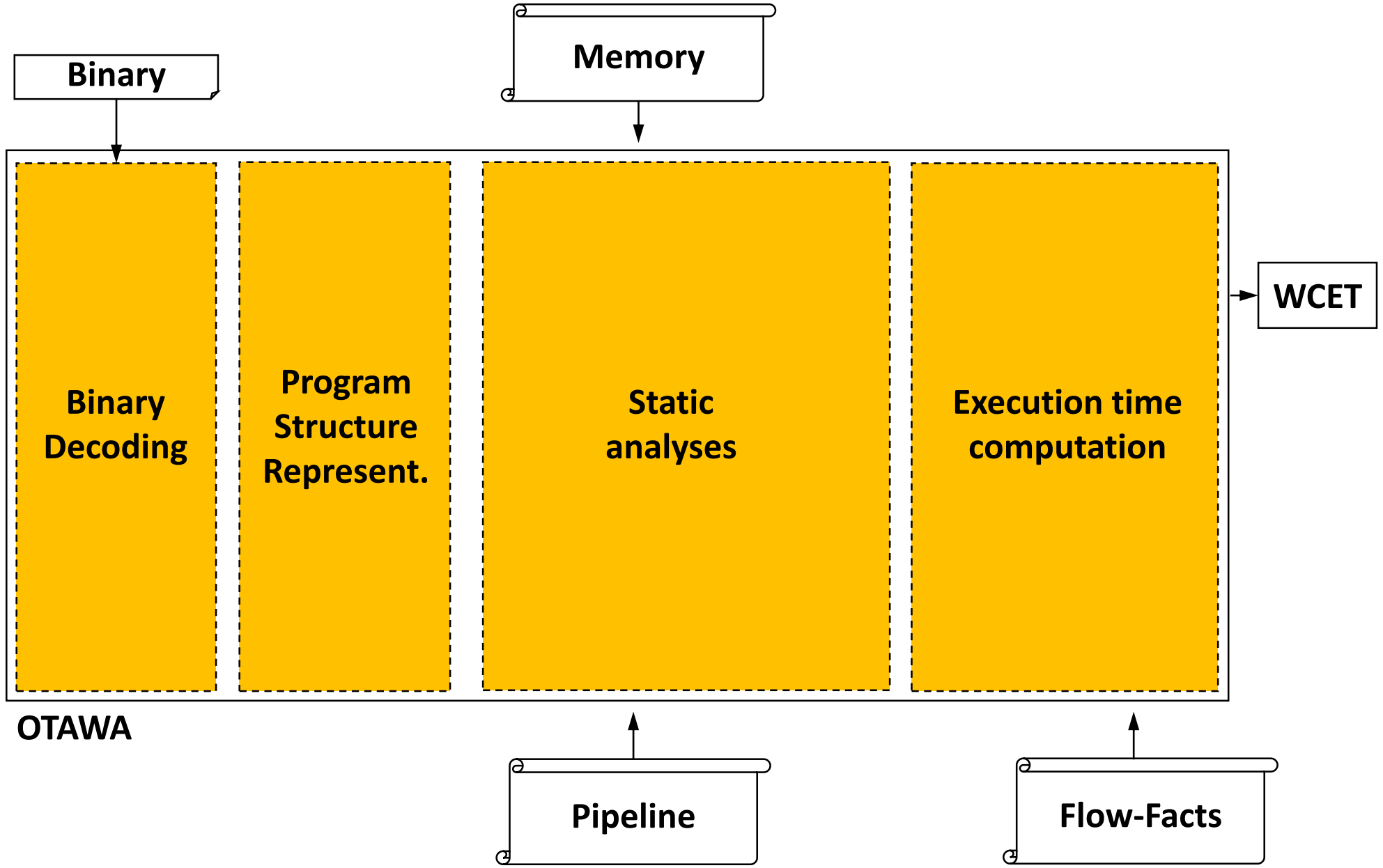
❖ 3rd step: perform the static analyses to capture hardware's effect



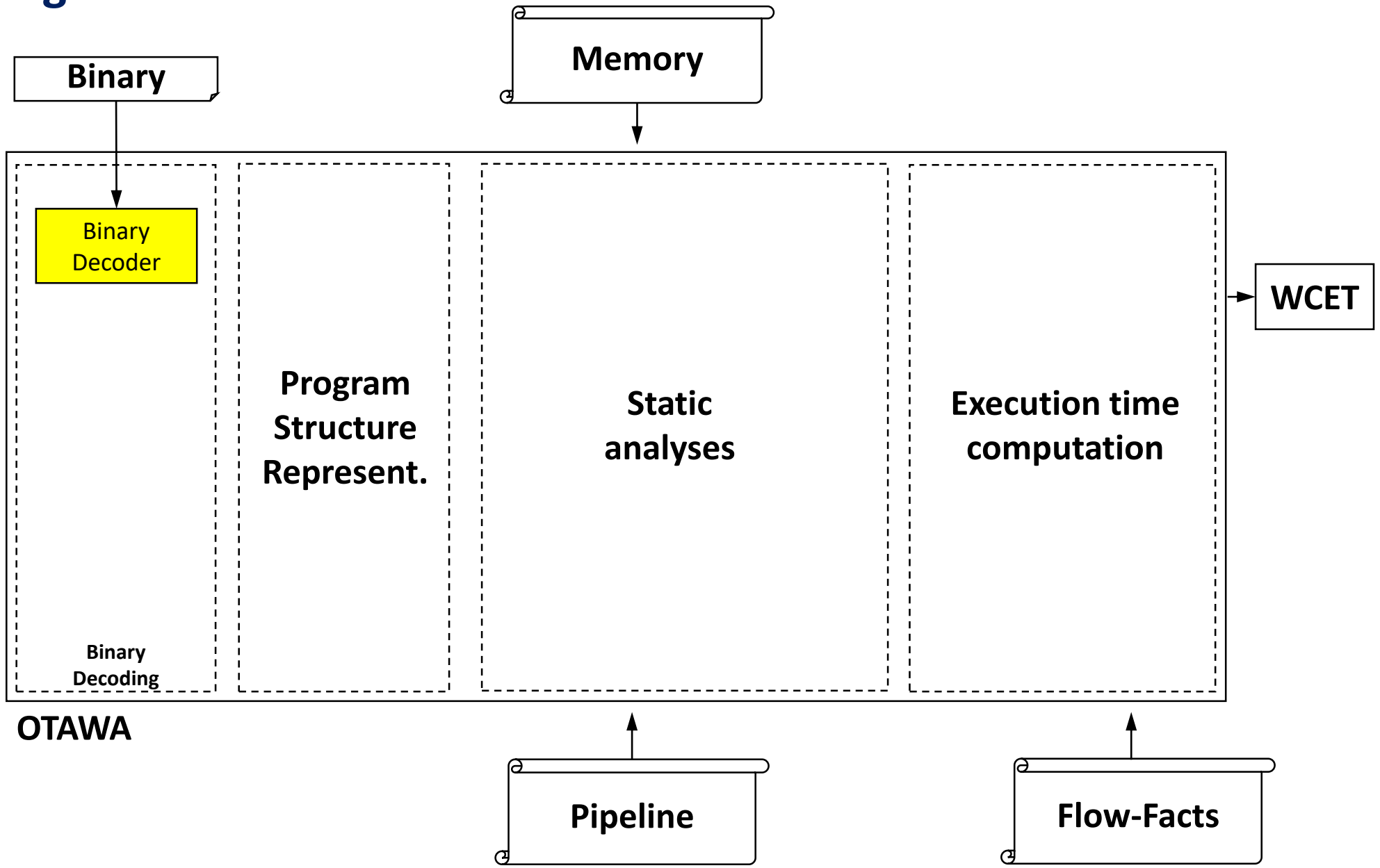
❖ 4th step: collect the analyses results and compute the time for instructions



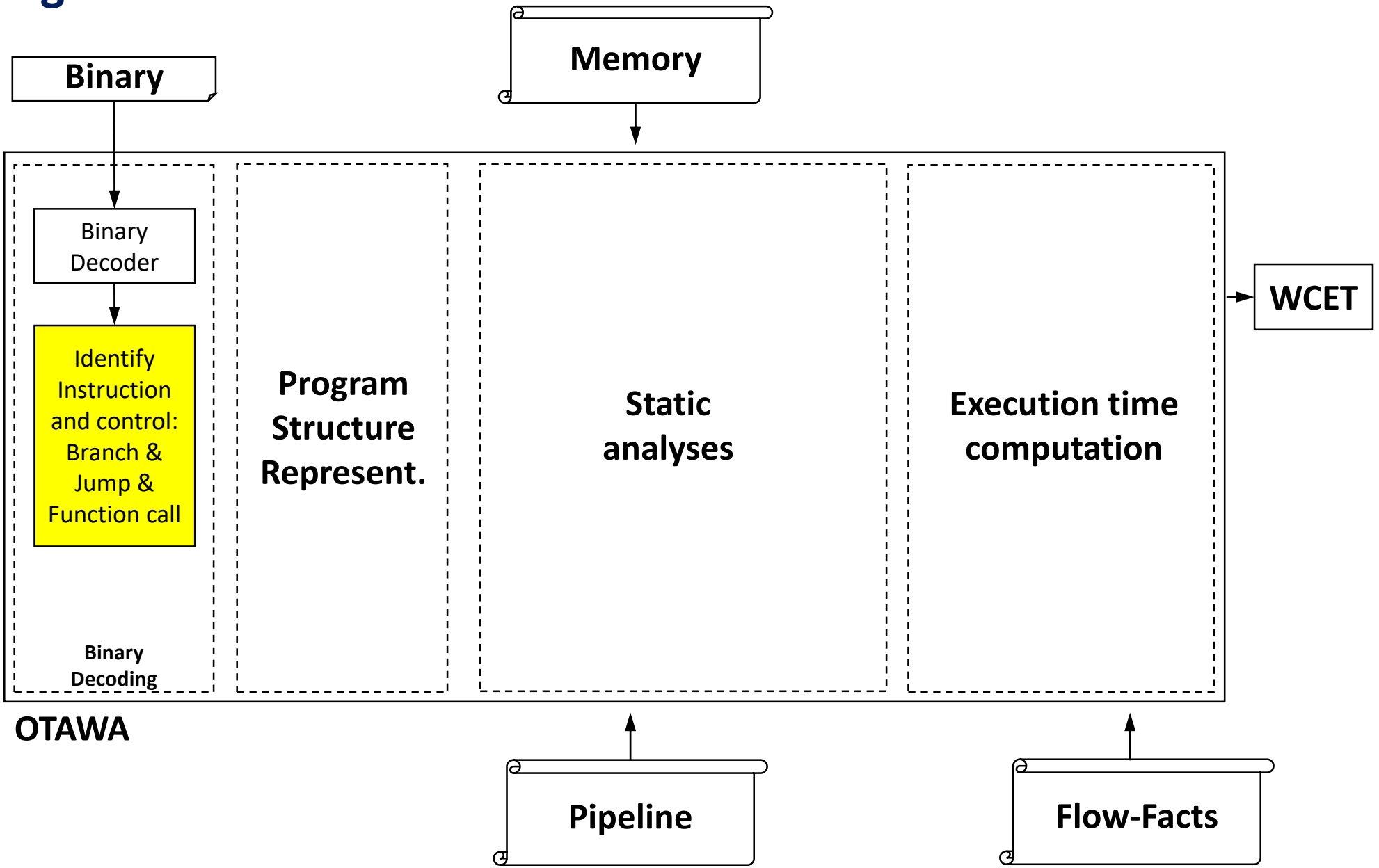
❖ The Golden Model of OTAWA



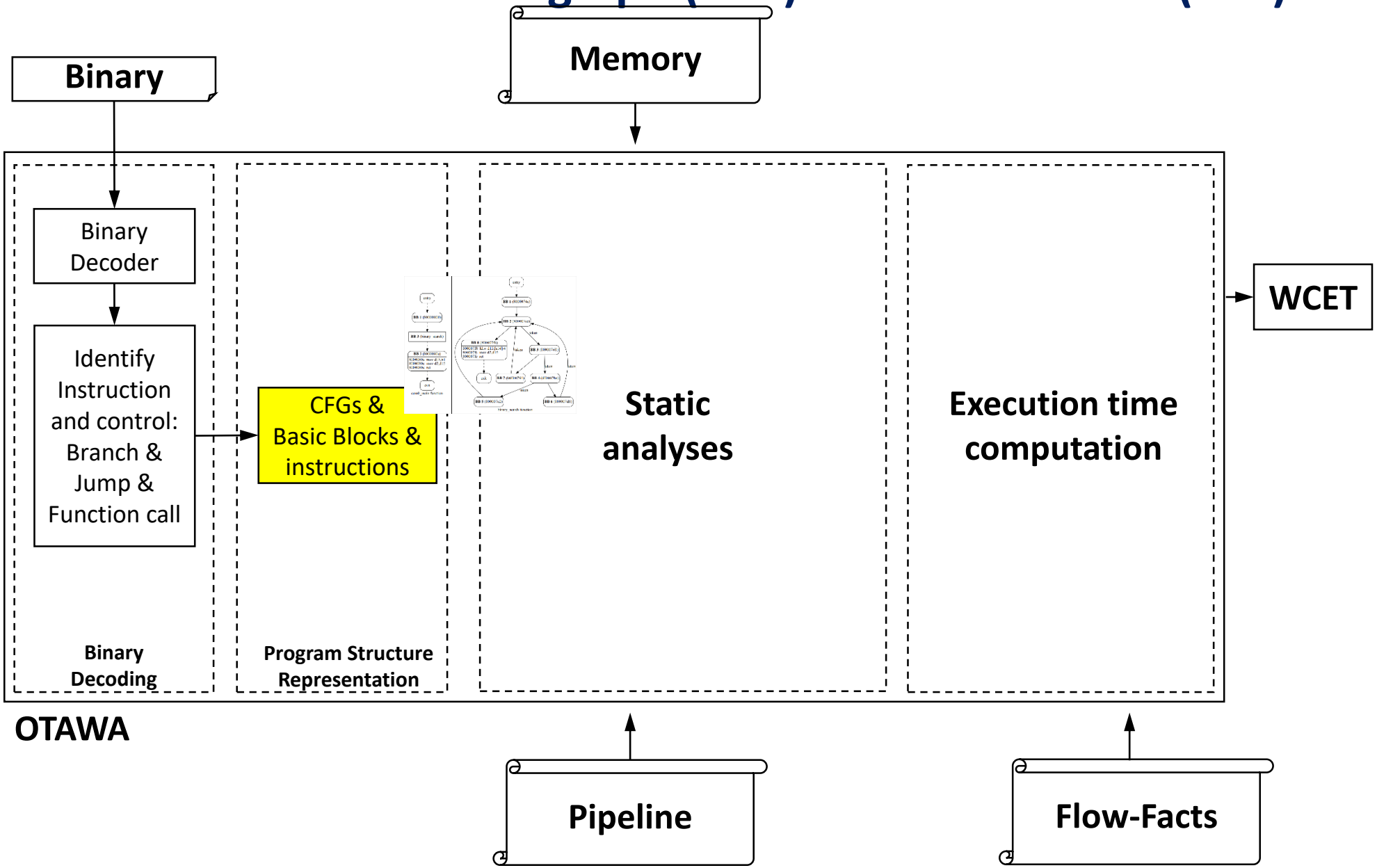
❖ Binary decoding - revisited



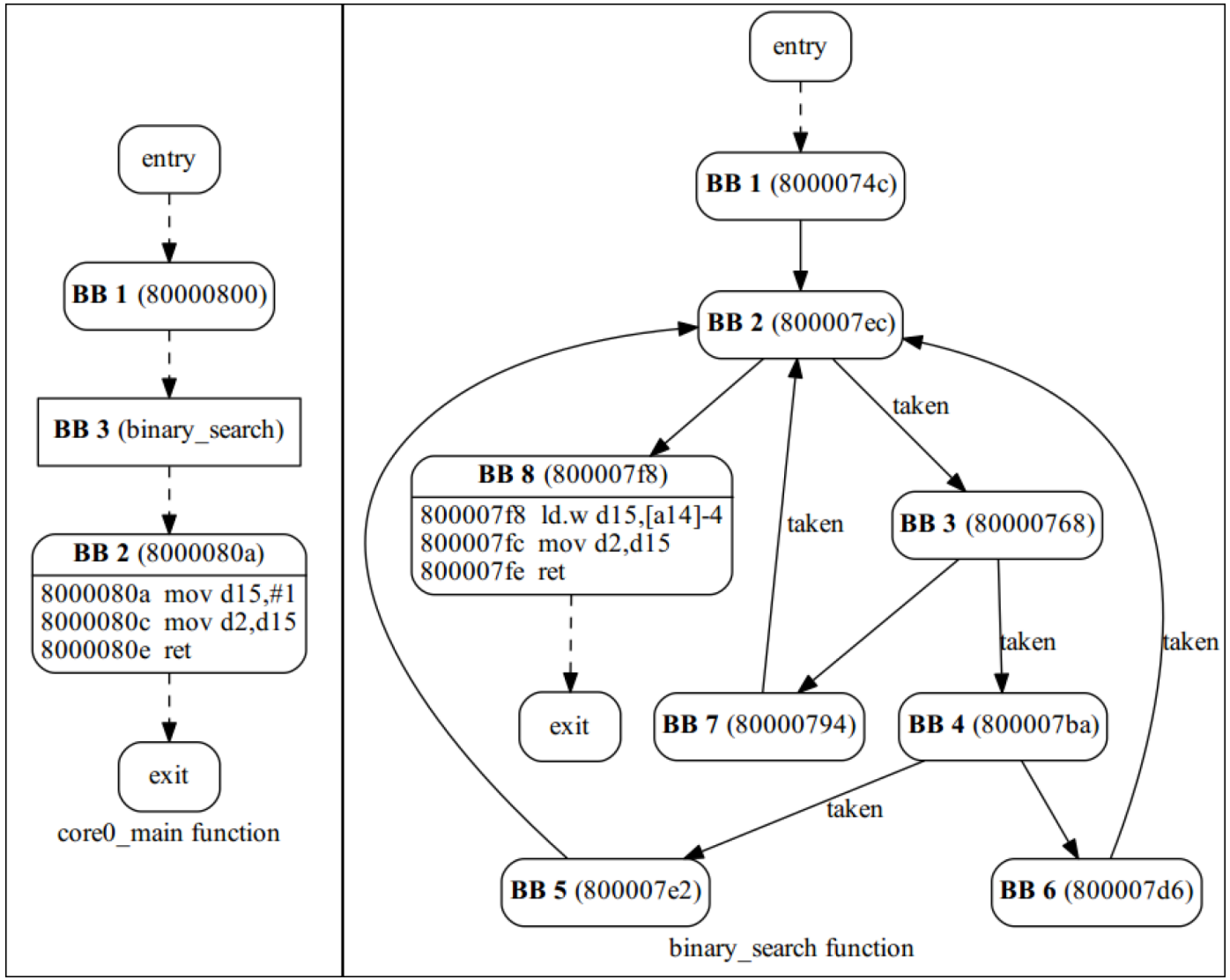
❖ Binary decoding - revisited



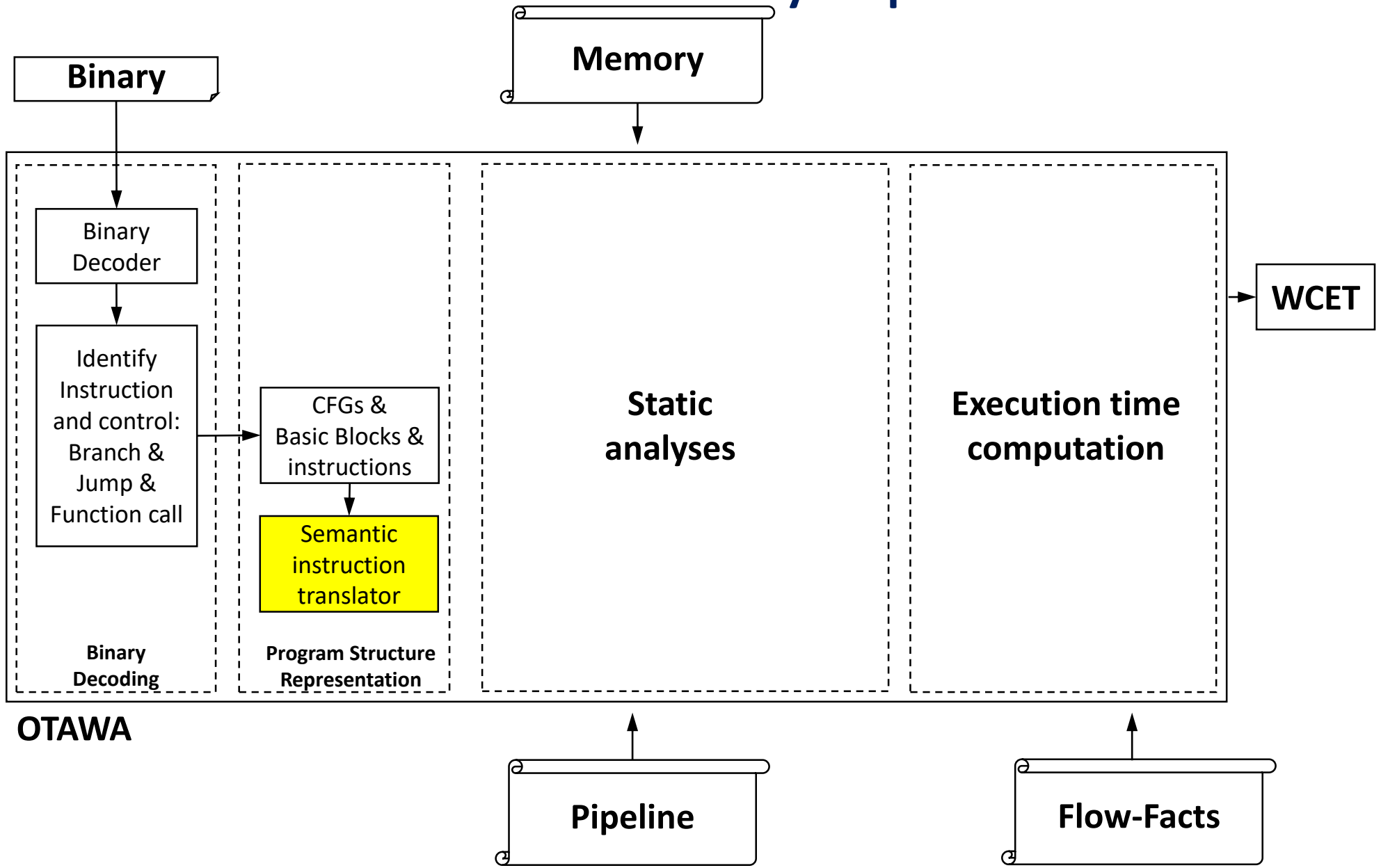
❖ Software representation: control-flow graph (CFG) and basic-blocks (BBs)



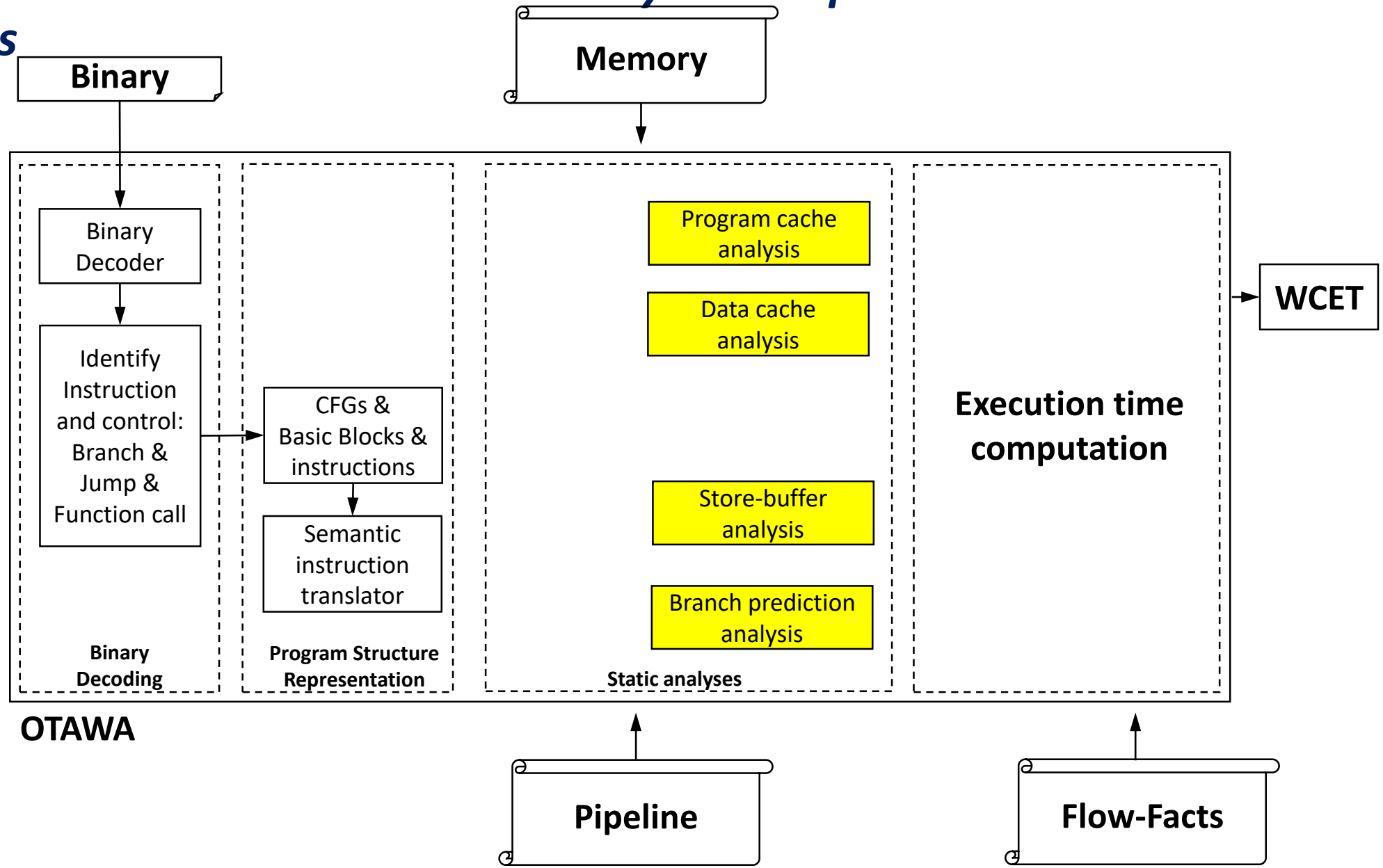
❖ Software representation: control-flow graph (CFG) and basic-blocks (BBs)



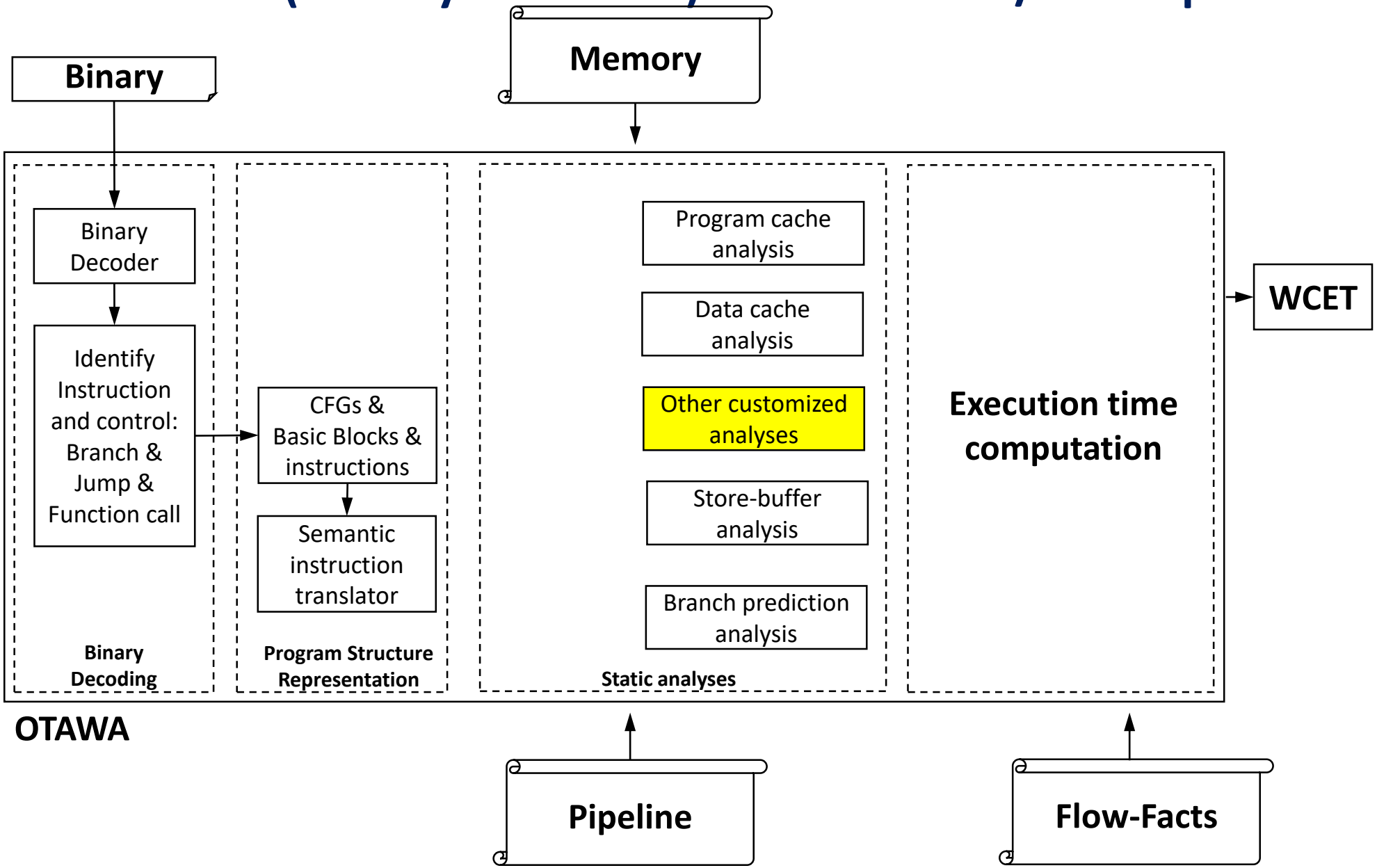
❖ Semantic instructions are used to make later analyses platform independent



❖ Now to consider hardware with *static analyses – capture the behaviours of components*

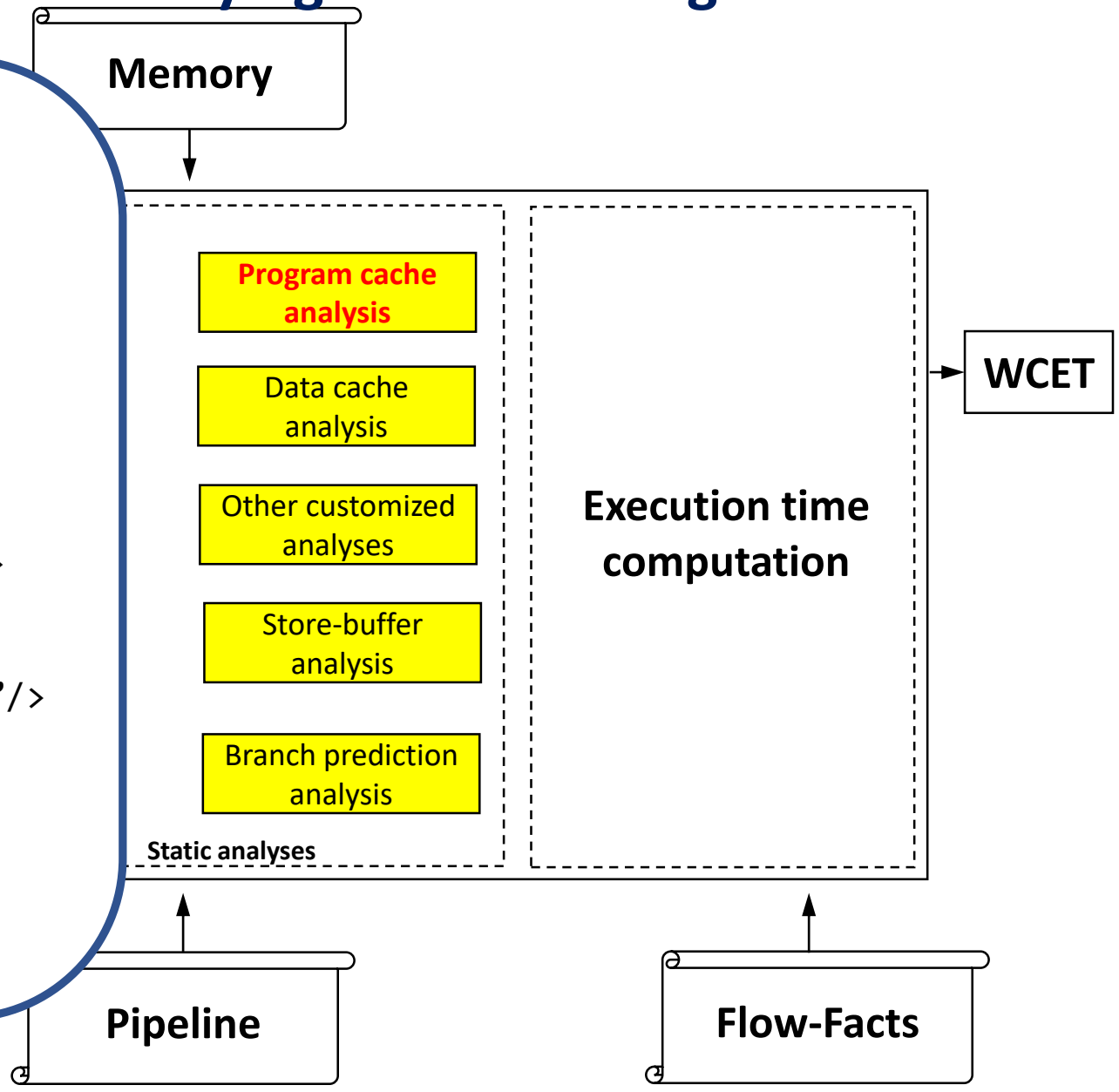


❖ Analyses could be built-in (already in OTAWA) or customized/developed



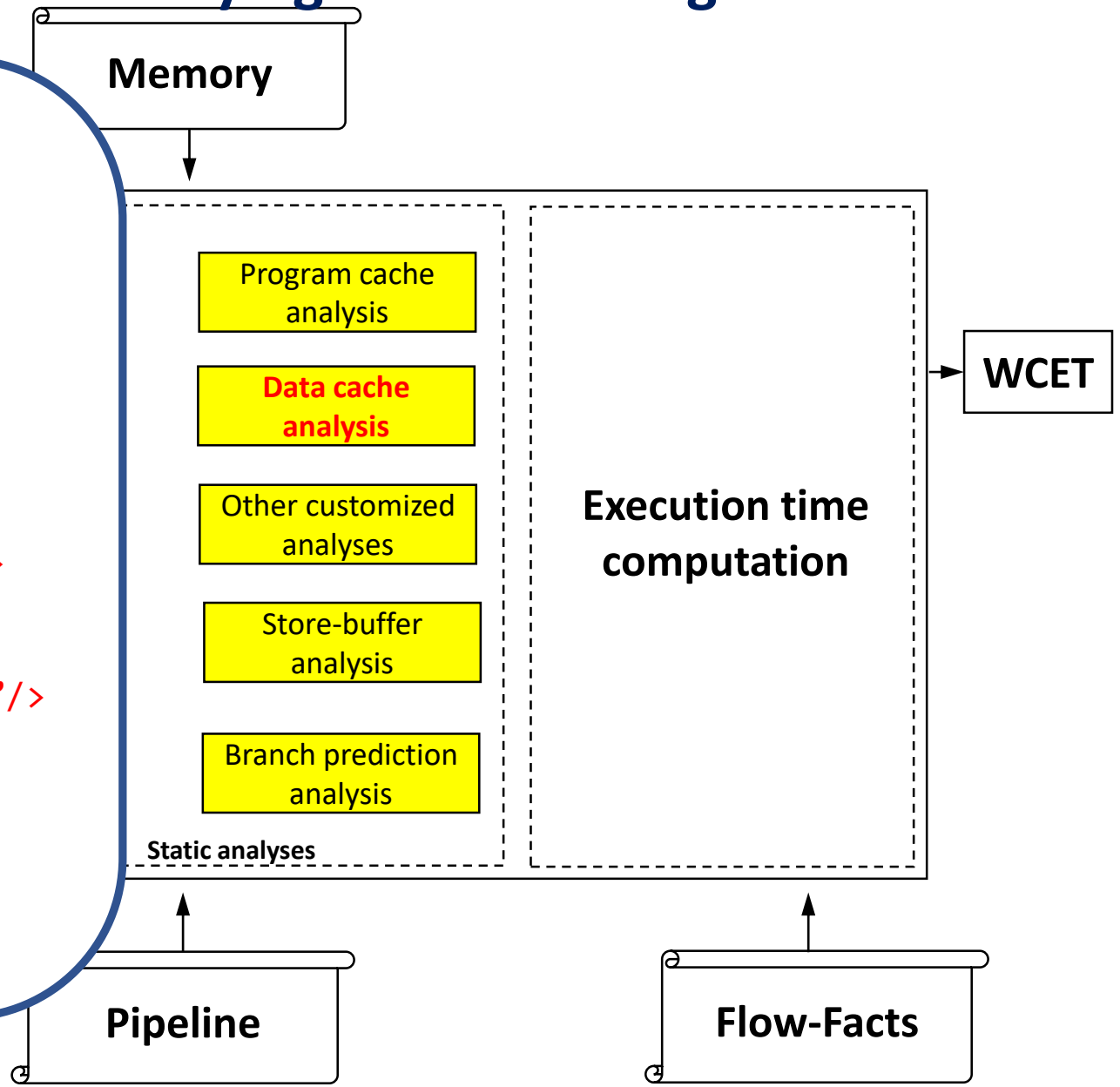
❖ Analysis are chosen according to the underlying hw and config via XML

```
<script>
 <step require =”tricore16::BranchPredTC16E”/>
 <step require=”otawa::ICACHECATEGORY2FEATURE”/>
 <step require=”otawa::ICACHEONLYCONSTRAINT2”/>
 <step require=”otawa::clp::CLPANALYSISFEATURE”/>
 <step require=”otawa::dcache::CLPBlockBuilder”/>
 <step require=”otawa::dcache::ACSMustPersBuild”/>
 <step require=”otawa::dcache::ACSMayBuilder”/>
 <step require=”otawa::dcache::CATBuilder”/>
 <step require=”otawa::dcache::CatConstraintBuild”/>
 ......
</script>
```



❖ Analysis are chosen according to the underlying hw and config via XML

```
<script>
 <step require =”tricore16::BranchPredTC16E”/>
 <step require=”otawa::ICACHECATEGORY2FEATURE”/>
 <step require=”otawa::ICACHEONLYCONSTRAINT2”/>
 <step require=”otawa::clp::CLPANALYSISFEATURE”/>
 <step require=”otawa::dcache::CLPBlockBuilder”/>
 <step require=”otawa::dcache::ACSMustPersBuild”/>
 <step require=”otawa::dcache::ACSMayBuilder”/>
 <step require=”otawa::dcache::CATBuilder”/>
 <step require=”otawa::dcache::CatConstraintBuild”/>
 ......
</script>
```

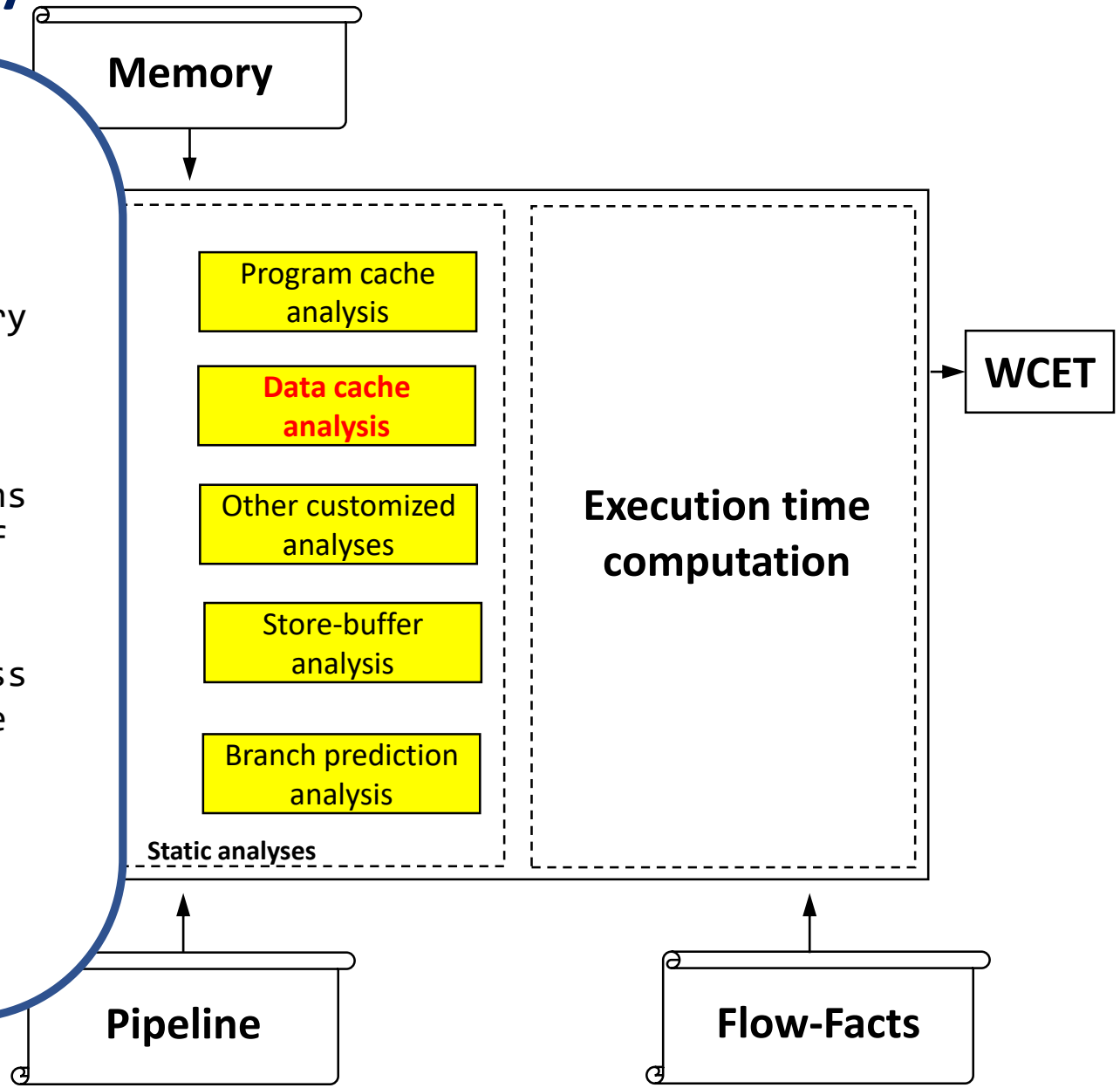


❖ Dependences existed between analyses

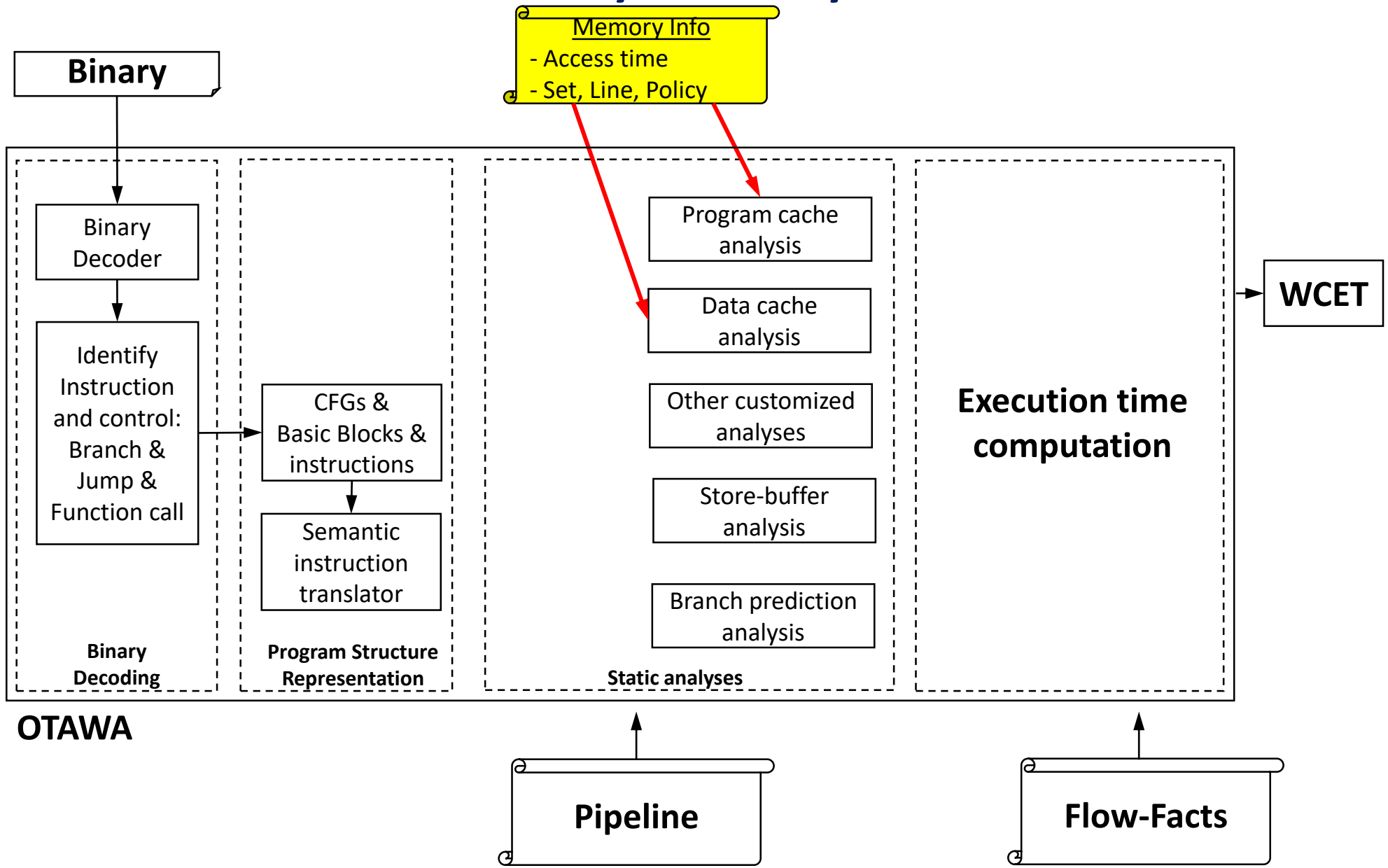
1. **Address and value analysis** (CLP) is used to determine the values of the registers and memory locations.

2. **Data cache analysis** is only **feasible** once the **accessed addresses are known** by the instructions such as LD (load) and ST (store). The states of the data cache can be derived.

3. **Category analysis** determines if a cache hit-miss occurs depending on the cache state (from above analysis) and the current access (from the CLP analysis)

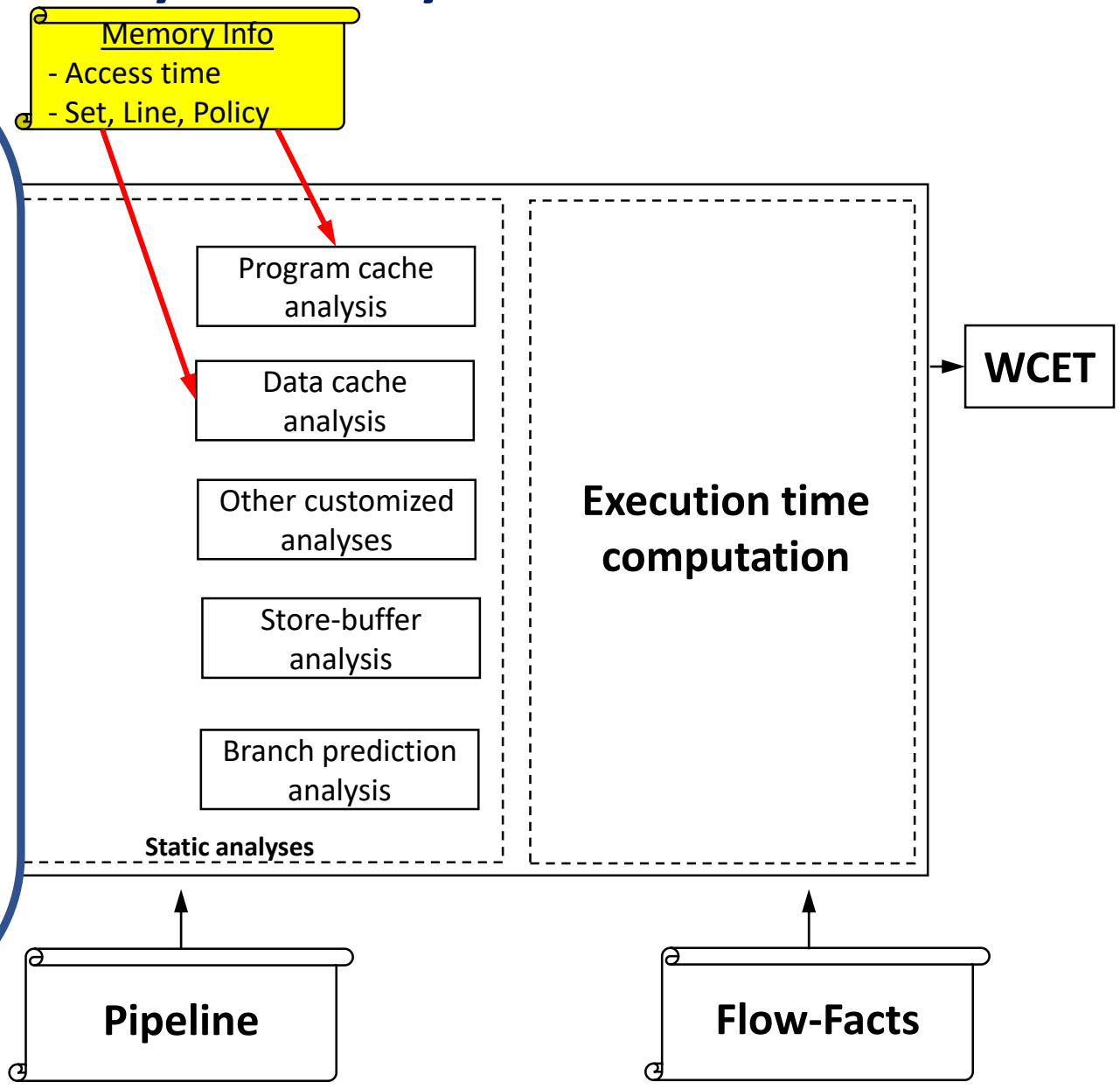


❖ Need to specified the hardware – memory hierarchy and characteristics



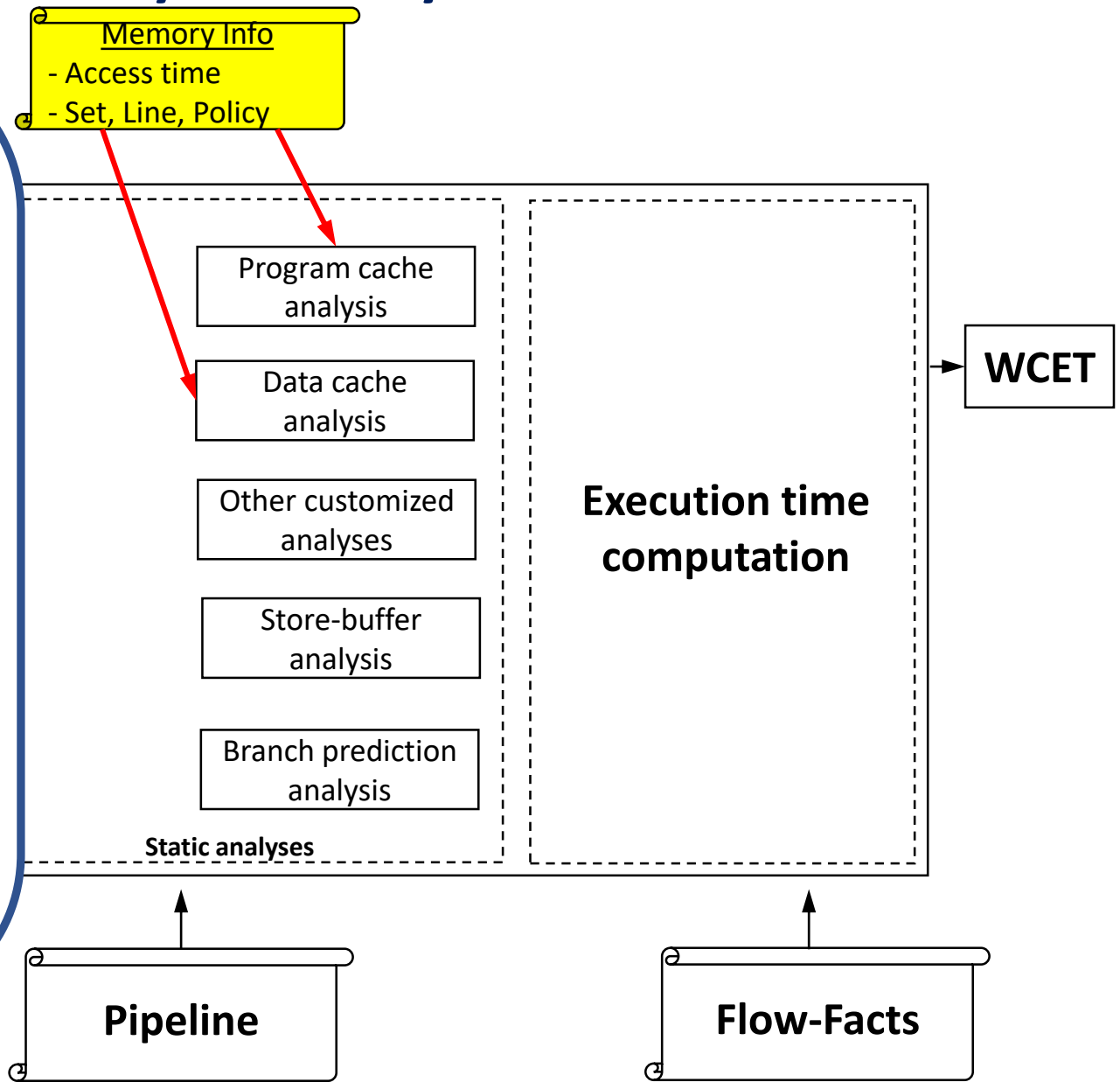
❖ Need to specified the hardware – memory hierarchy and characteristics

```
<memory>
  <bank>
    <name>PFLASH0</name>
    <address>0x80000000</address>
    <size>0x01000000</size><!--2MBytes/-->
    <latency>13</latency>
    <writable>false</writable>
    <cacheable>true</cacheable>
    </bank><bank>...</bank>
</memory>
```

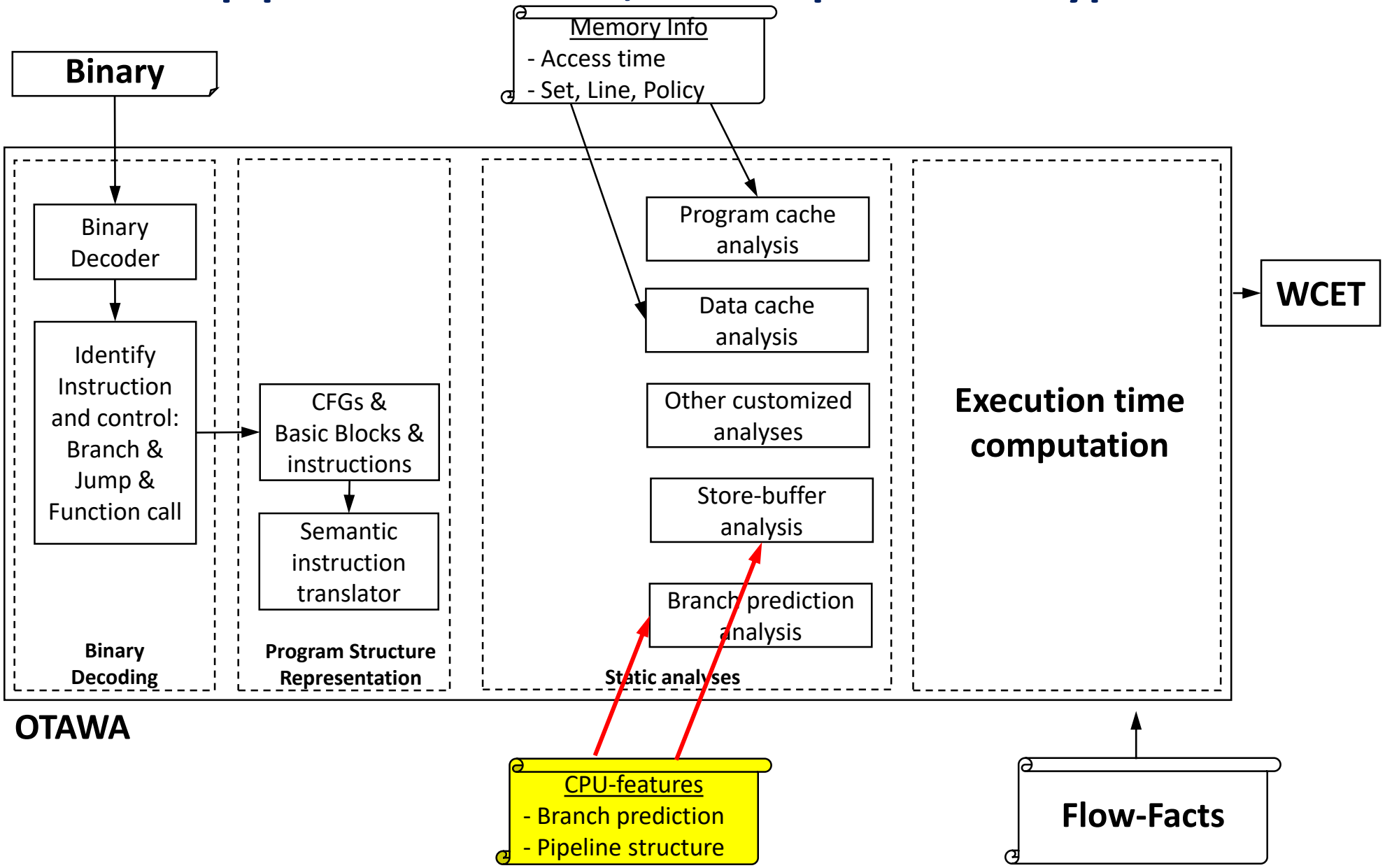


❖ Need to specified the hardware – memory hierarchy and characteristics

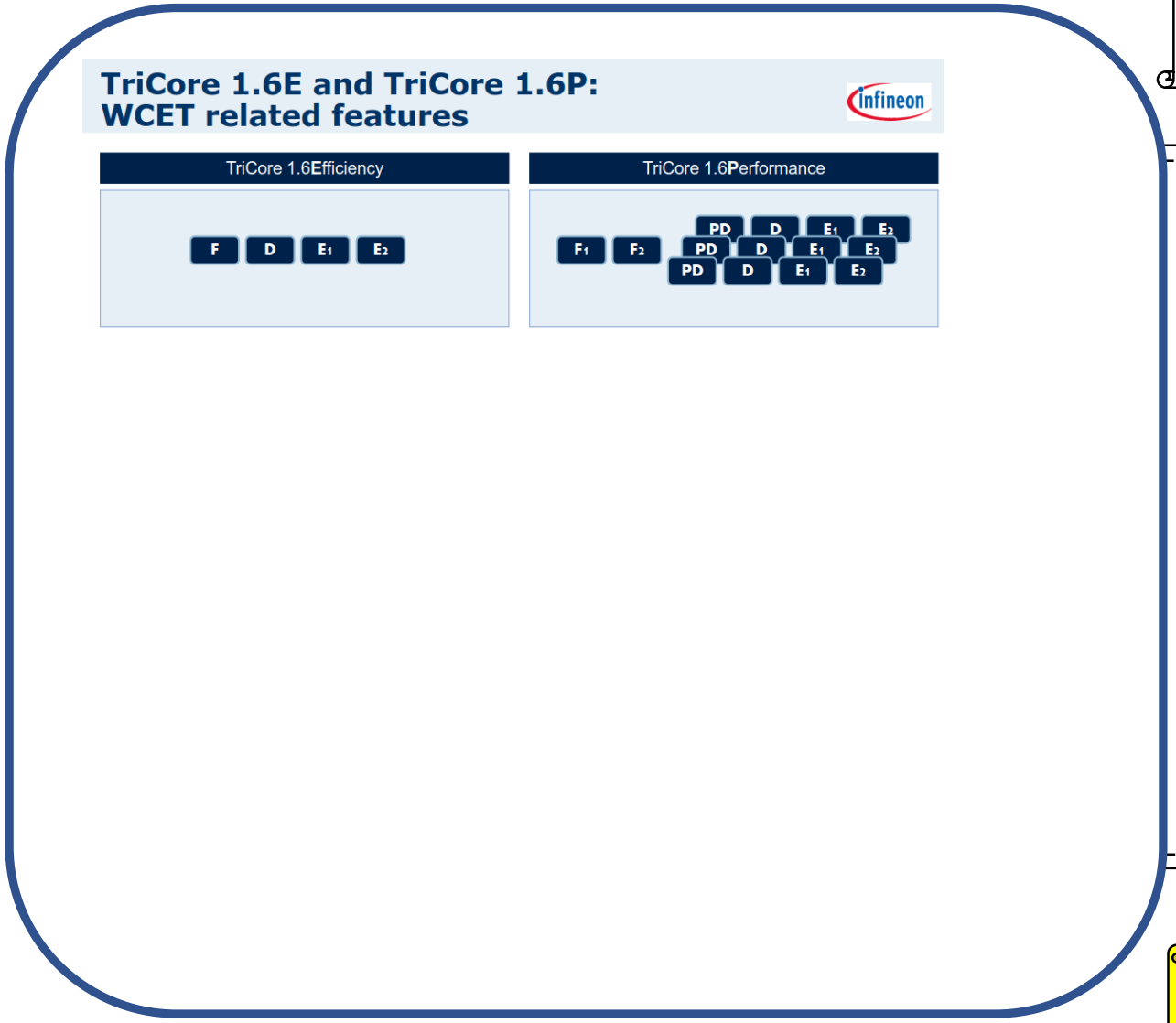
```
<memory>
  <bank>
    <name>PFLASH0</name>
    <address>0x80000000</address>
    <size>0x01000000</size><!--2MBytes/-->
    <latency>13</latency>
    <writable>false</writable>
    <cacheable>true</cacheable>
    </bank><bank>...</bank>
</memory>
<cache-config>
  <icache>
    <blockbits>5</blockbits
    <waybits>1</waybits>
    <rowbits>7</rowbits>
    </icache>
  <dcache>...</dcache>
</cache-config>
```

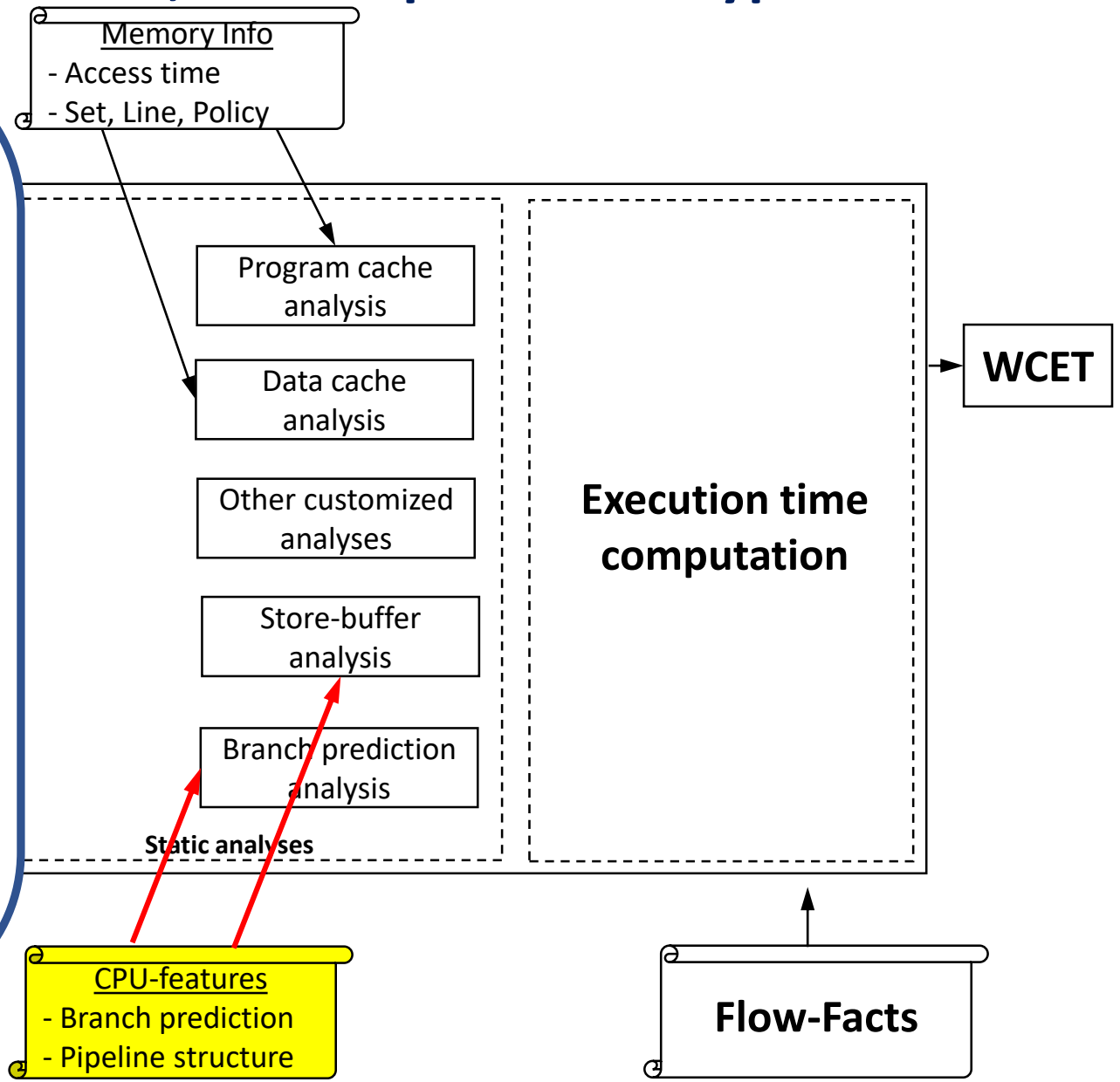


❖ CPU specific features – pipelines structure, branch prediction type

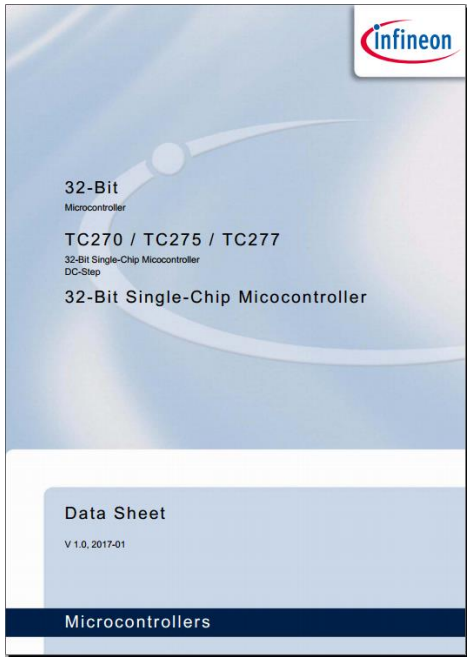


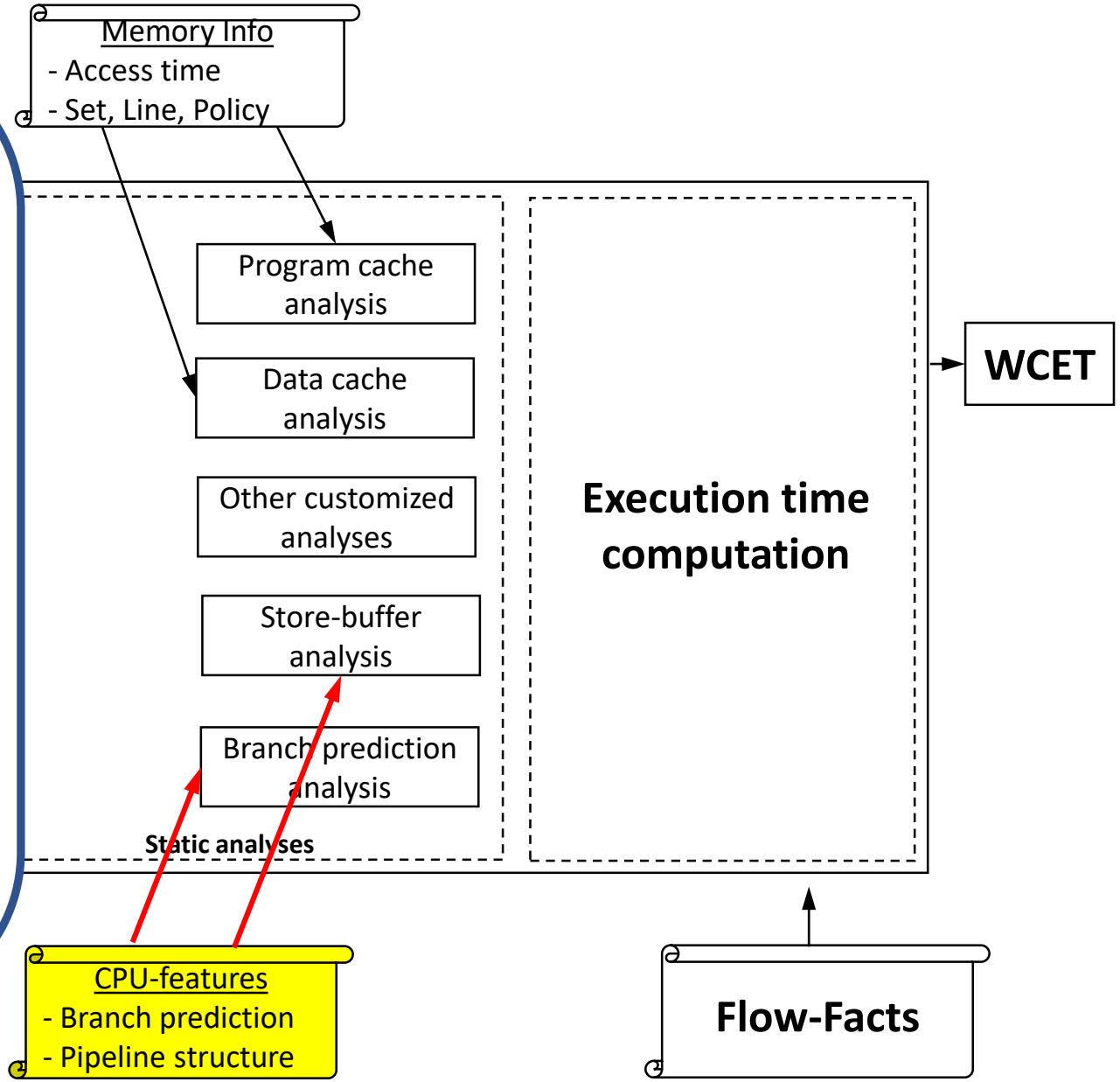
❖ CPU specific features – pipelines structure, branch prediction type



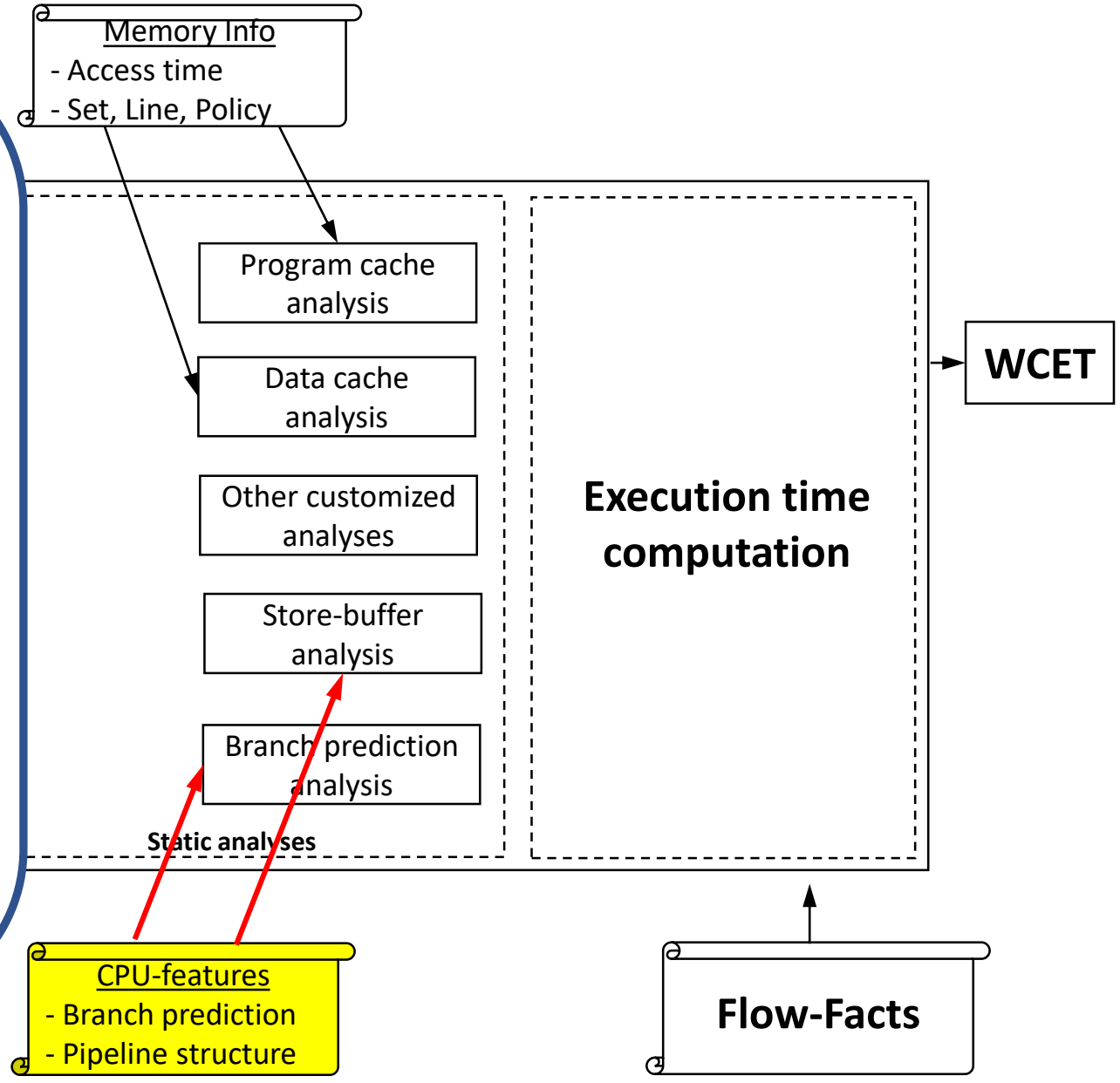


❖ CPU specific features – obtained from user-manuals, slides, data-sheets





❖ CPU specific features – obtained from user-manuals, slides, data-sheets

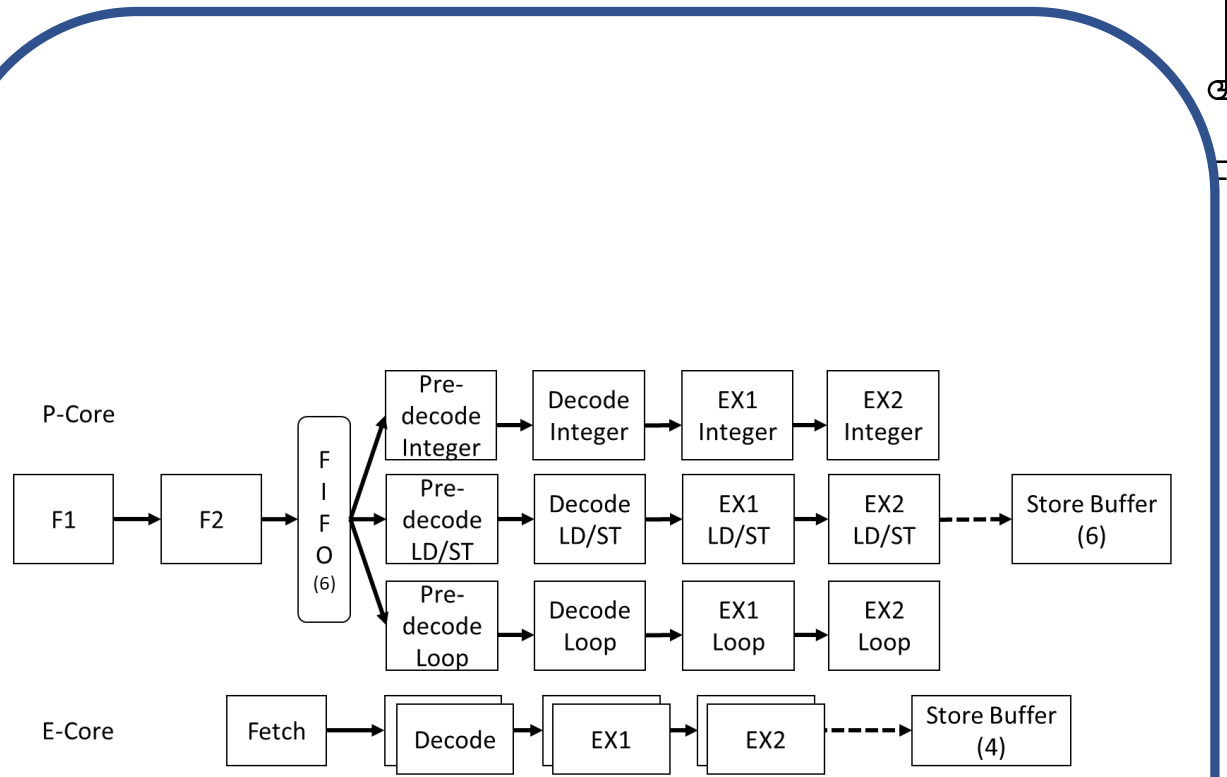


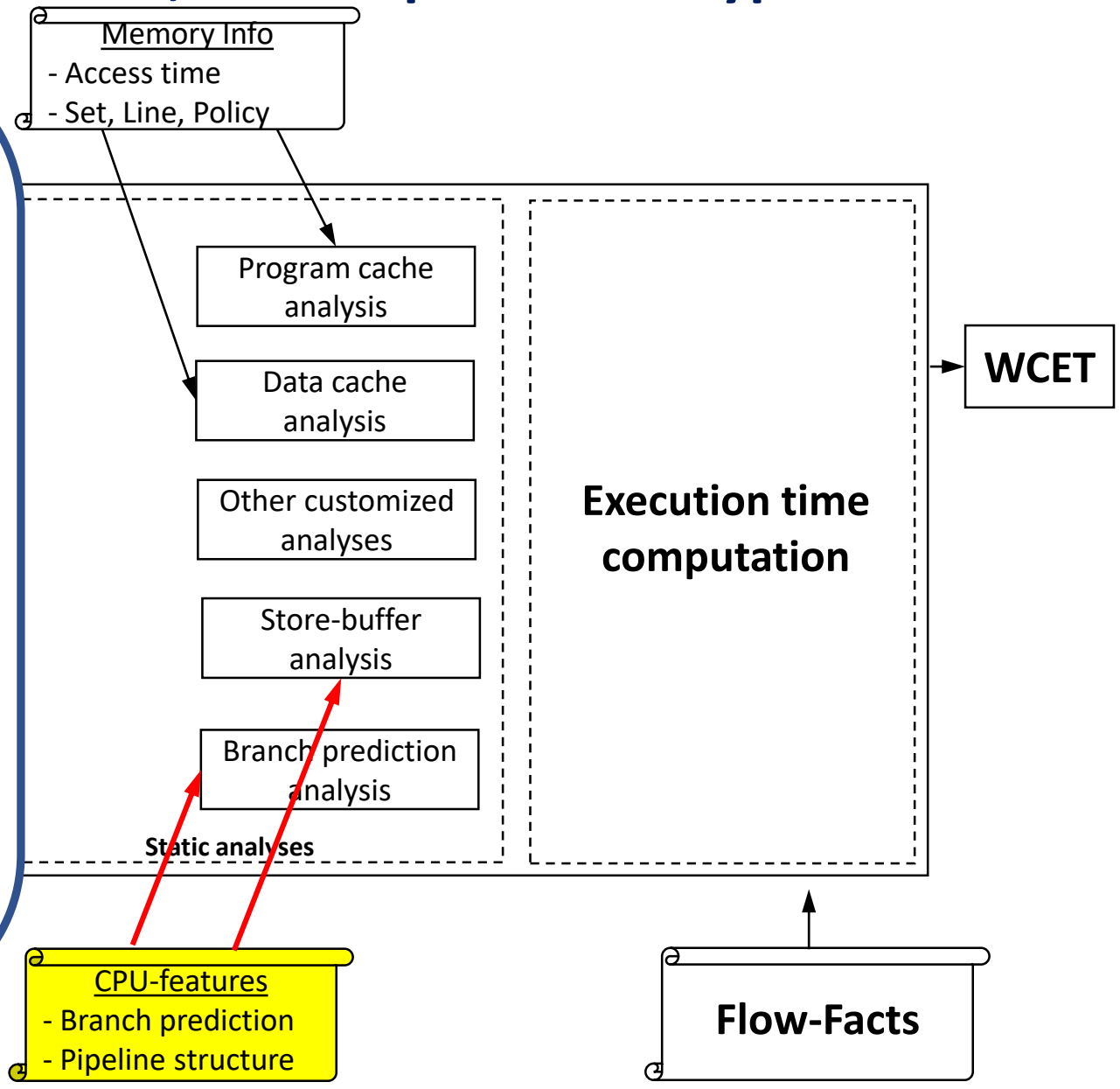
❖ Experiments on sequences of instructions to find hardware characteristics

❖ Use of performance counter (inst, cycles, hits/misses, stalls, multi-fetches)

Seq	Address	Instruction | cycle	-m-5	-m-4	-m-3	-m-2	-m-1	-m	-n-1	-n	-4	-3	-2	-1	0	1
-3	0x800011b4	st.w [a6]536 <f8700218>,d3	0	PD.M	DE.M	E1.M	E2.M									
-2	0x800011b8	ld.w d3,[a6]536 <f8700218>	1	0	PD.M	DE.M	E1.M	E2.M…	E2.M							
-1	0x800011bc	isync	Stop Fetch	1	0	-	-	-	-							
0	0x800011c0	mtcr 0xfc00,d15								Fetch…	0	PD.M	DE.M	E1.M	E2.M	
1	0x800011c4	mtcr 0xfc00,d2									1	0	PD.M	DE.M	E1.M	E2.M
2	0x800011c8	mfcr d2,core_id										1	0	PD.M	DE.M	-
3	0x800011cc	mfcr d1,0xfc04										2	1	0	PD.M	-
4	0x800011d0	mfcr d2,0xfc08											2	1	0	0
5	0x800011d4	mfcr d3,0xfc0c											3	2	1	1
6	0x800011d8	mfcr d4,0xfc10												3	2	2
7	0x800011dc	mfcr d5,0xfc14												4	3	3
8	0x800011e0	extr.u d1,d1,0,31												Fetch	-	-
	1 instructions	1 cycles														
	1	PMEM_STALL														1
	0	DMEM_STALL														
	0	PCACHE_HIT														
	0	PCACHE_MISS														
	0	IP_STALL														
	1	LS_STALL														1
	0	MULTI_ISSUE														
	0	DCACHE_HIT														
	0	DCACHE_MISS														
	0x800011C0	SRI_ACCESS								x						

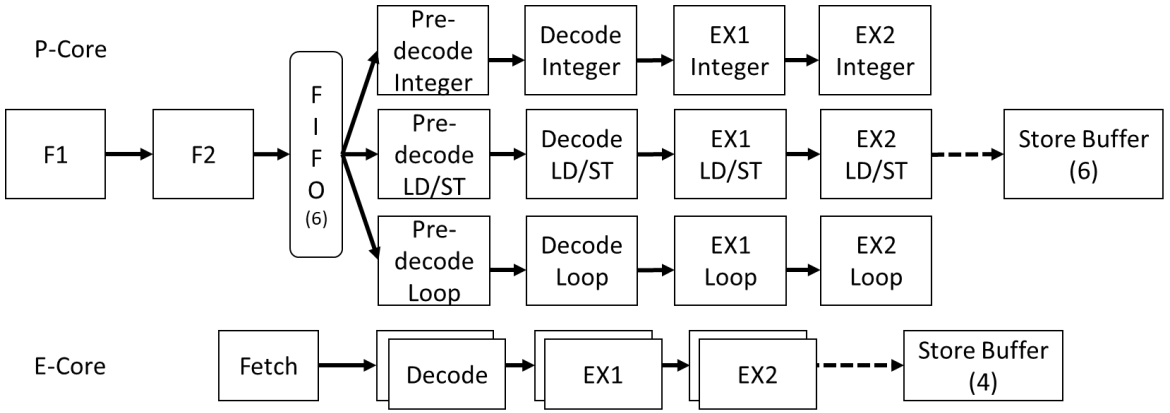
❖ CPU specific features – pipelines structure, branch prediction type

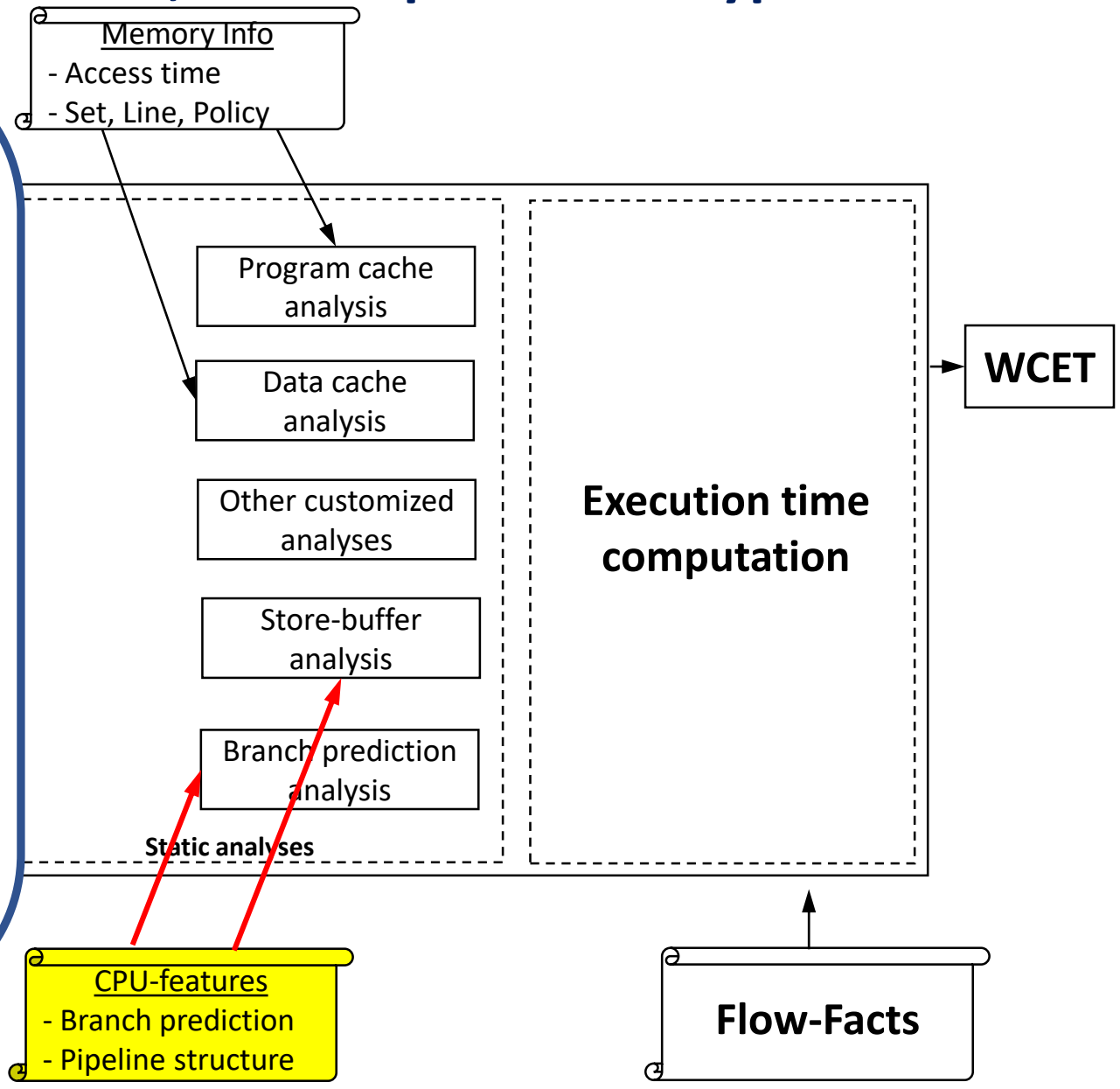




❖ CPU specific features – pipelines structure, branch prediction type

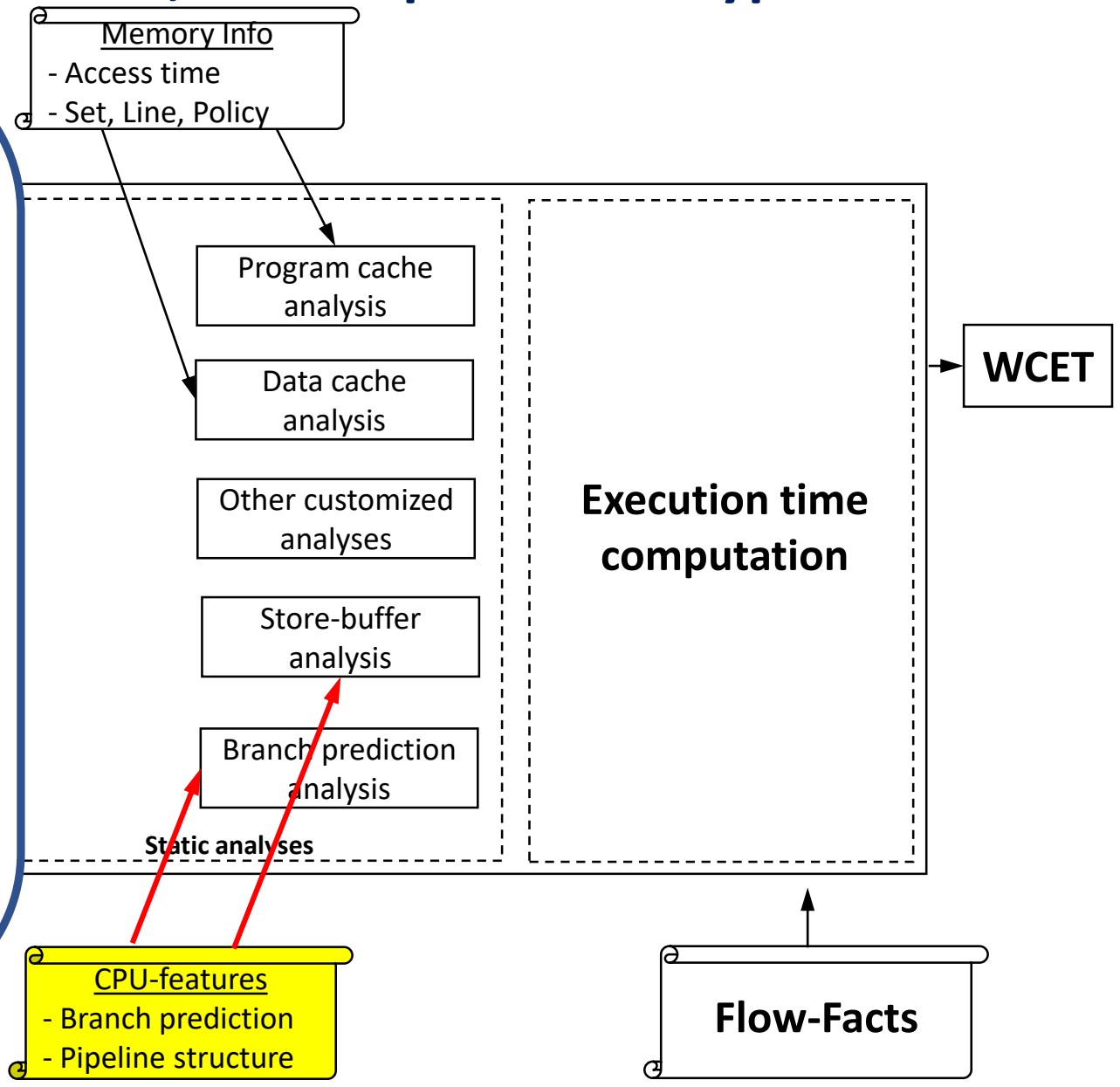
```
<stageid=”F1”><type>FETCH</type></stage>
<stageid=”F2”><type>LAZY</type></stage>
<stageid=”PD”><type>LAZY</type></stage>
<stageid=”DE”><type>LAZY</type></stage>
<stageid=”EXE”><type>EXEC</type>
```





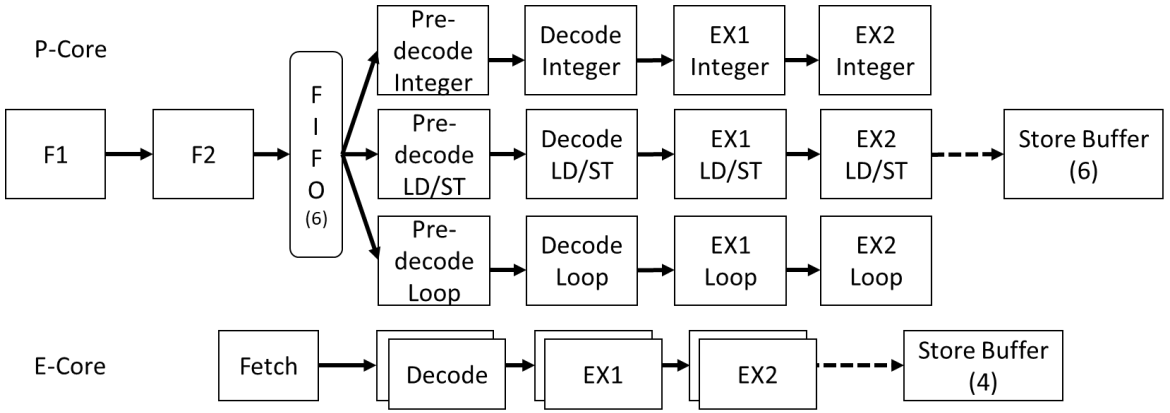
❖ CPU specific features – pipelines structure, branch prediction type

```
<stageid=”F1”><type>FETCH</type></stage>
<stageid=”F2”><type>LAZY</type></stage>
<stageid=”PD”><type>LAZY</type></stage>
<stageid=”DE”><type>LAZY</type></stage>
<stageid=”EXE”><type>EXEC</type>
<ordered>true</ordered>
<!--Thepipelines-->
<fuid=”EXEL”><latency>2</latency></fu>
<fuid=”EXEI”><latency>2</latency></fu>
<fuid=”EXEM”><latency>2</latency>
<mem>true</mem>
  <memstage>1</memstage>
</fu>
```

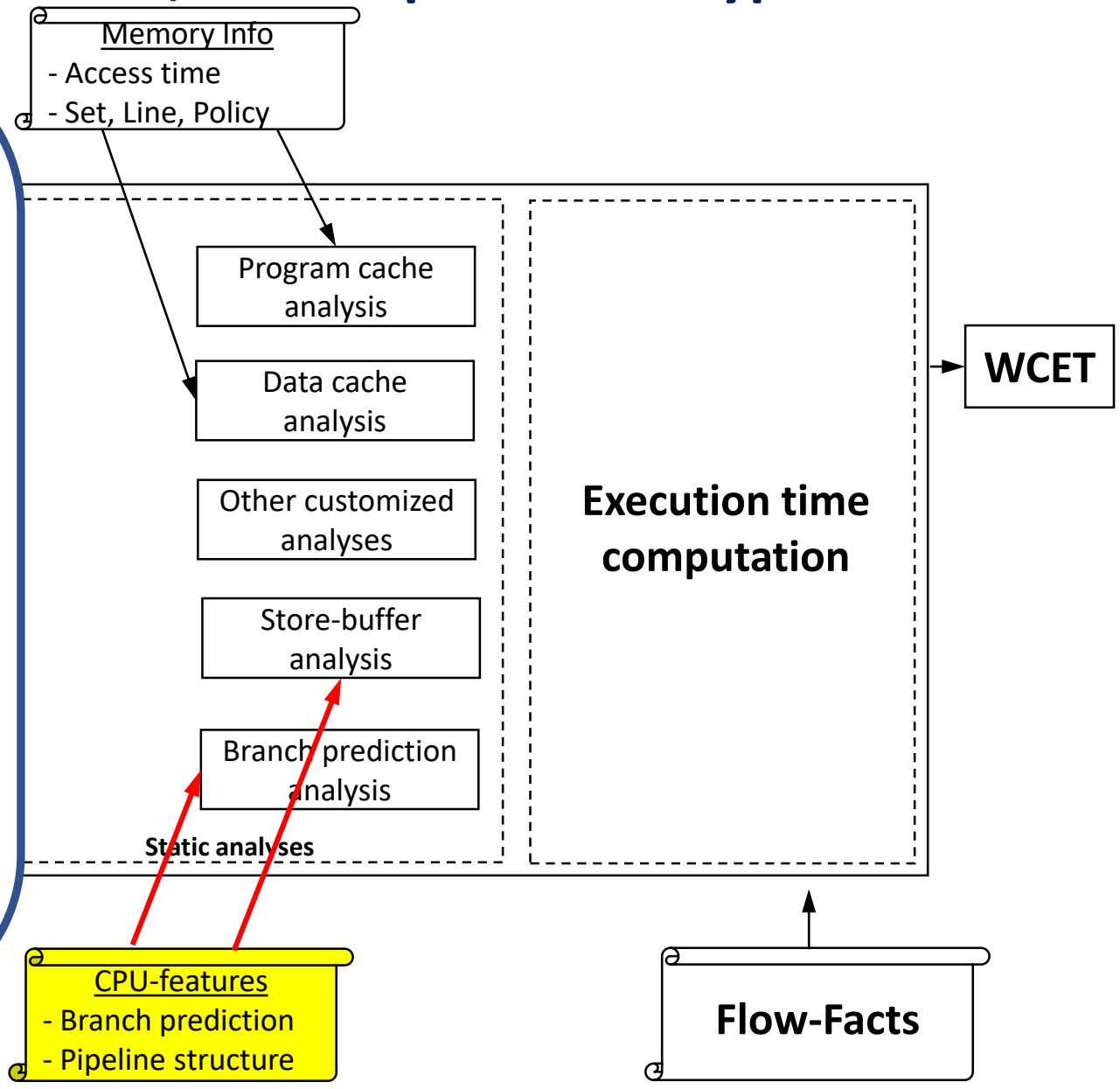


❖ CPU specific features – pipelines structure, branch prediction type

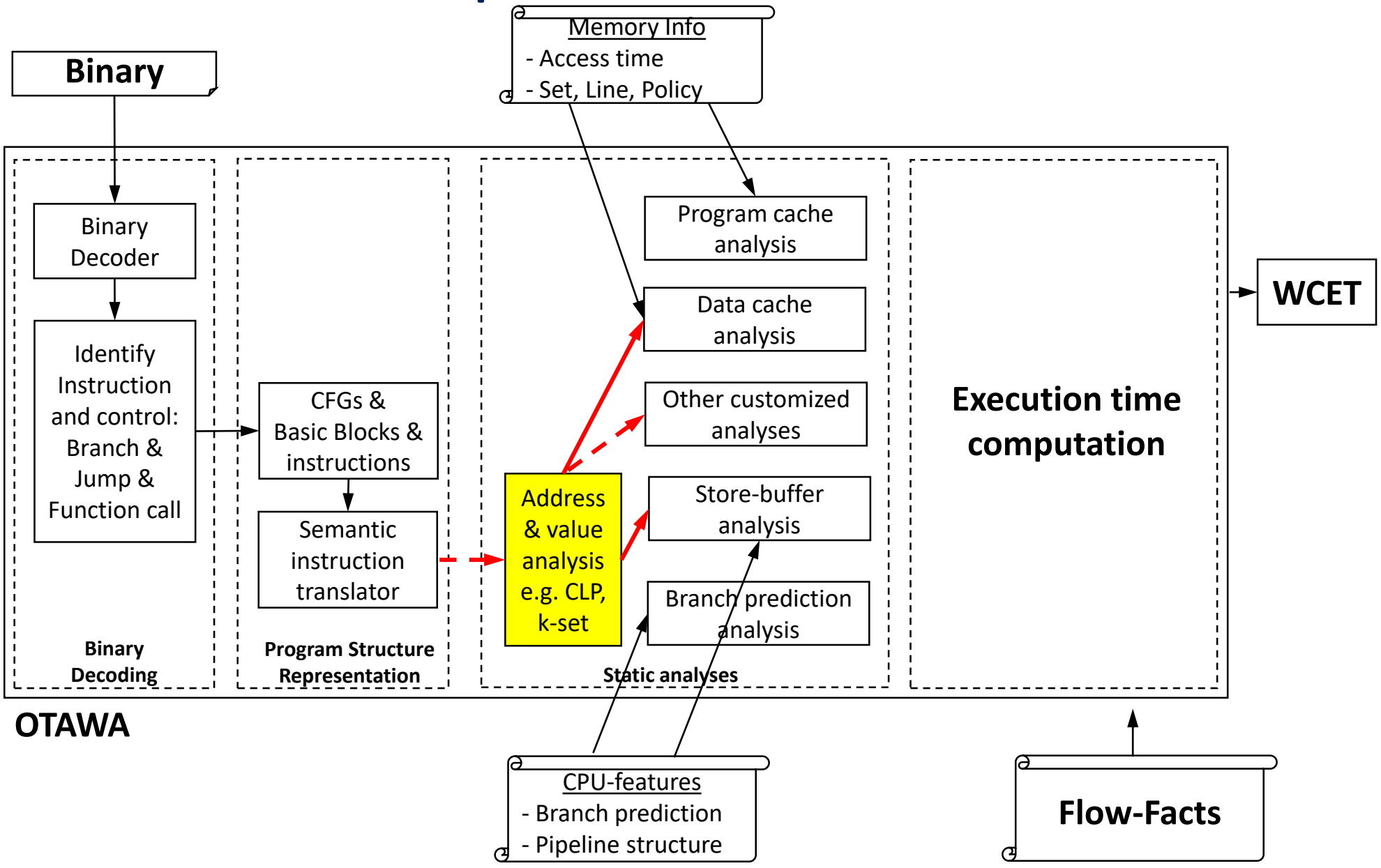
```
<stageid=”F1”><type>FETCH</type></stage>
<stageid=”F2”><type>LAZY</type></stage>
<stageid=”PD”><type>LAZY</type></stage>
<stageid=”DE”><type>LAZY</type></stage>
<stageid=”EXE”><type>EXEC</type>
```



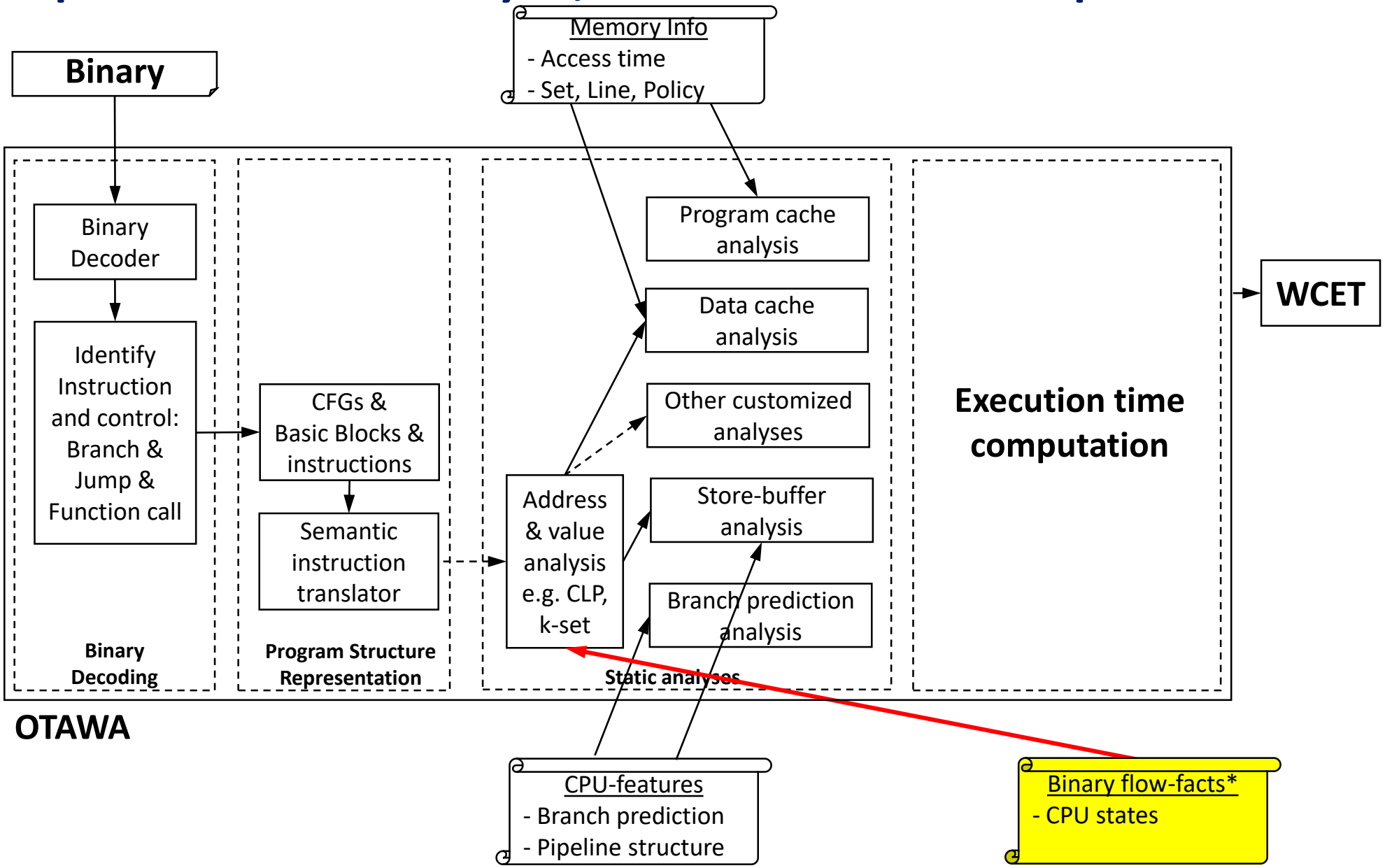

```
<dispatch>
  <type>0x80000000</type><furef=”EXEL”/>
  <type>0x40000000</type><furef=”EXEM”/>
  <type>0x10000000</type><furef=”EXEI”/>
</dispatch>
```



❖ The behaviours of the hardware depends on how software uses them

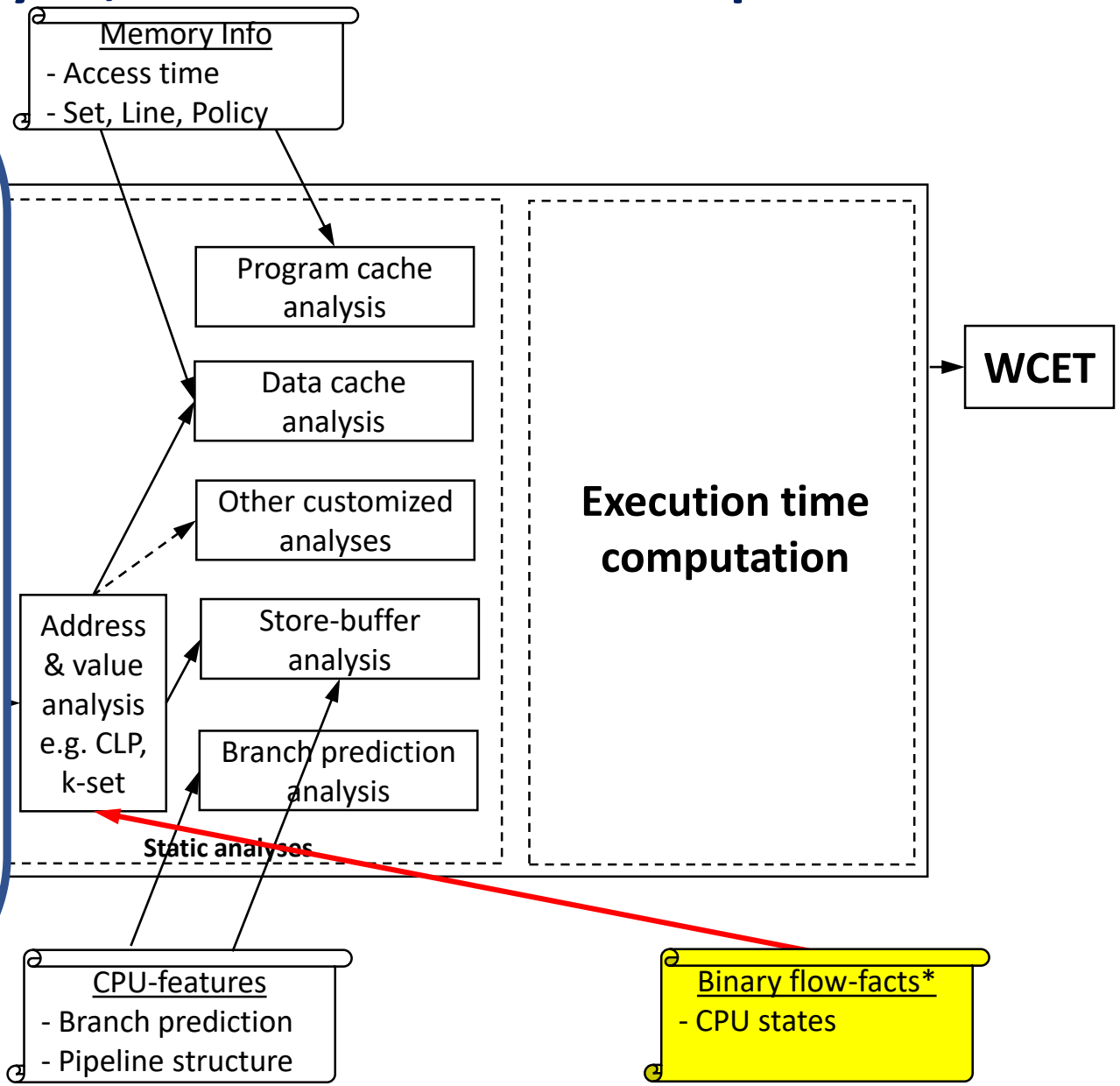


❖ To increase the precision of the analysis, CPU state can also be provided

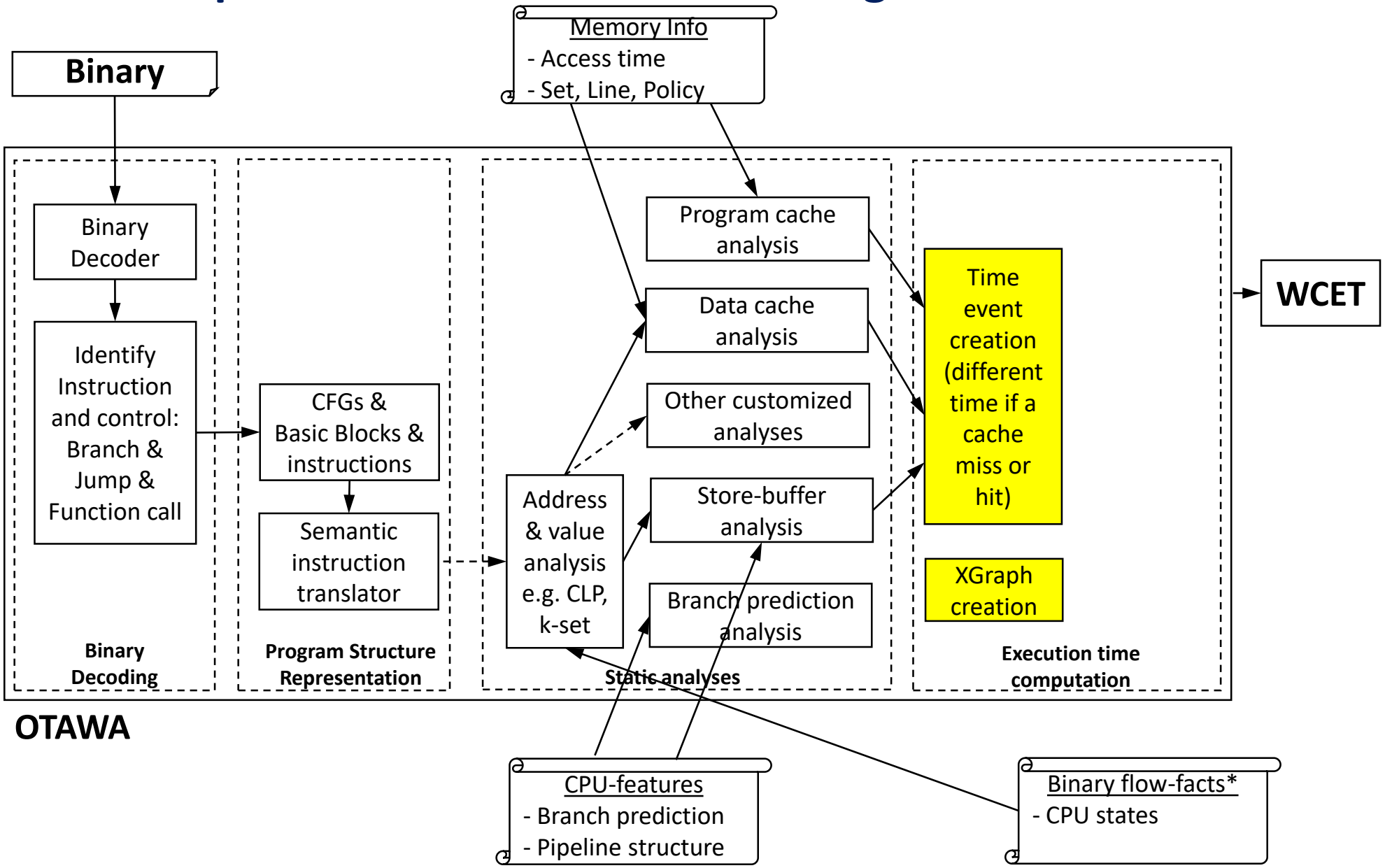


❖ To increase the precision of the analysis, CPU state can also be provided

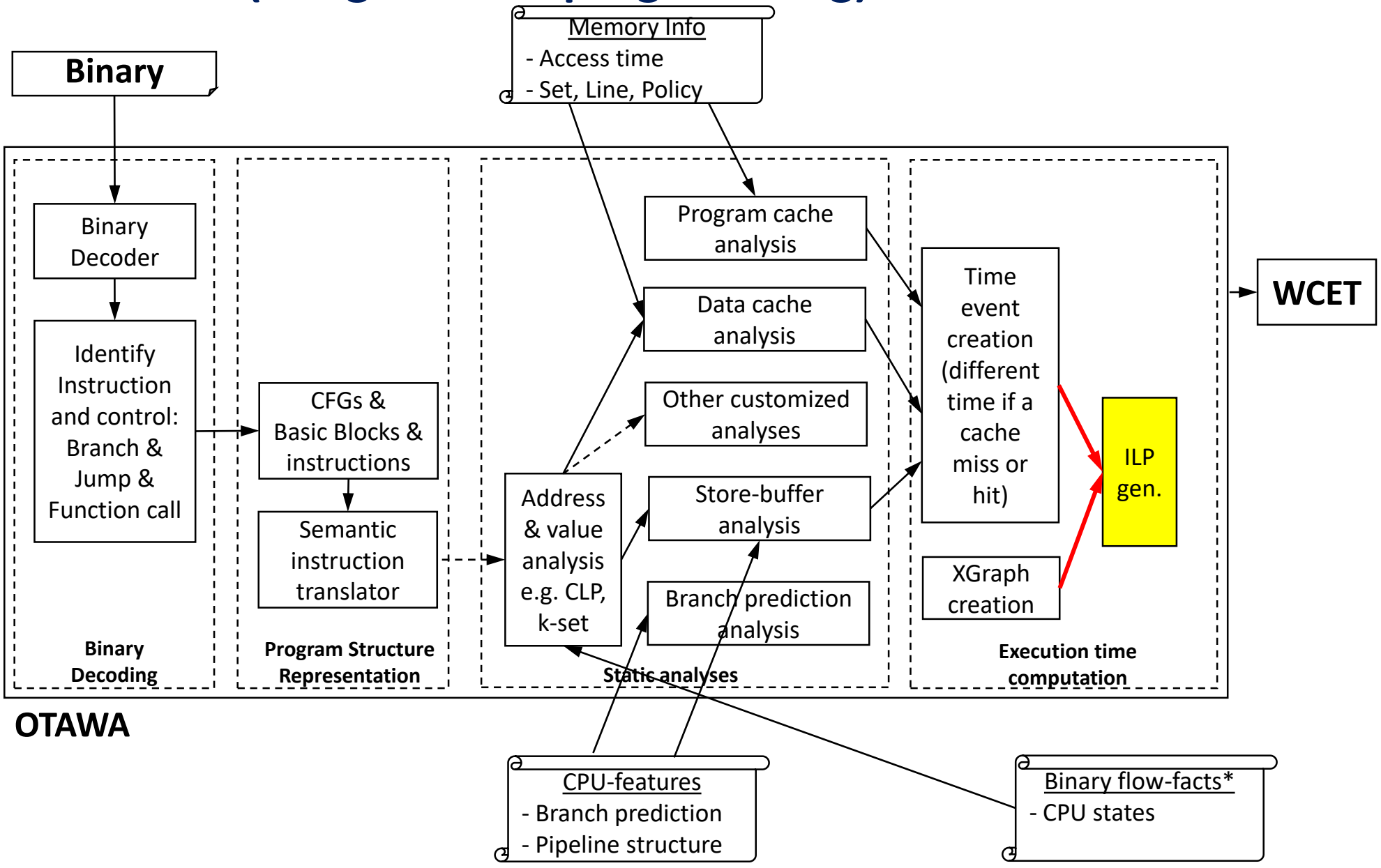
```
<!– Initial value --!>
<reg-init name = ”A10”
          value= ”0x70019600”/>
<reg-init name = ”PCXI”
          value= ”0x70019C00”/>
<mem-init address= ”0xD0000040”
          value= ”0x70019C00”/>

<!– Invariant value --!>
<state address=”0x80000402”>
  <reg name= ”A4”
       start=”0x600”step=”1”count=”400”/>
  <mem address=”0xD0000080”
       value=”0x70019C00”/>
</state>
```



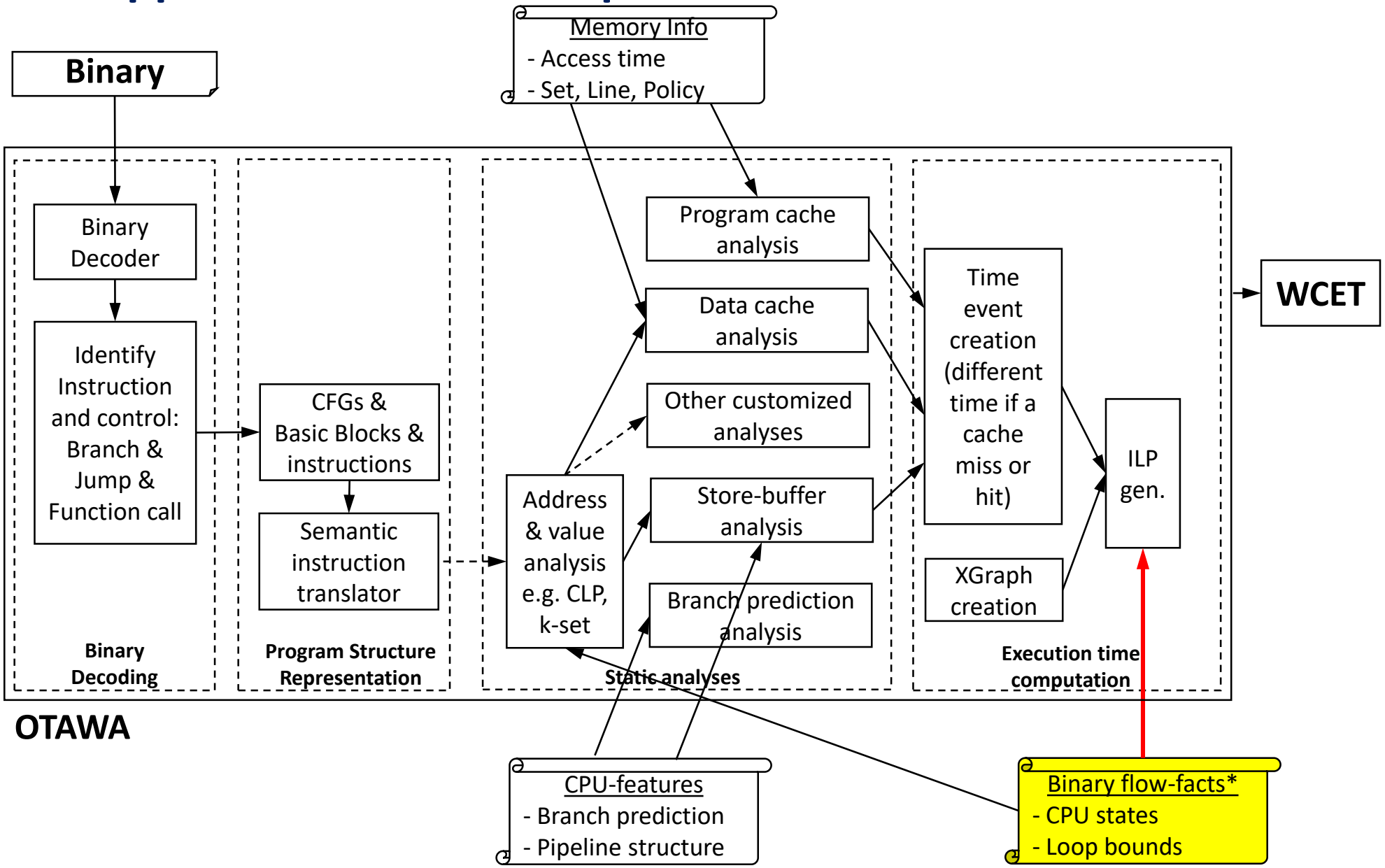
❖ Integrate the software paths and hardware effects together



❖ Put all variables into ILP (integer linear programming) formulae

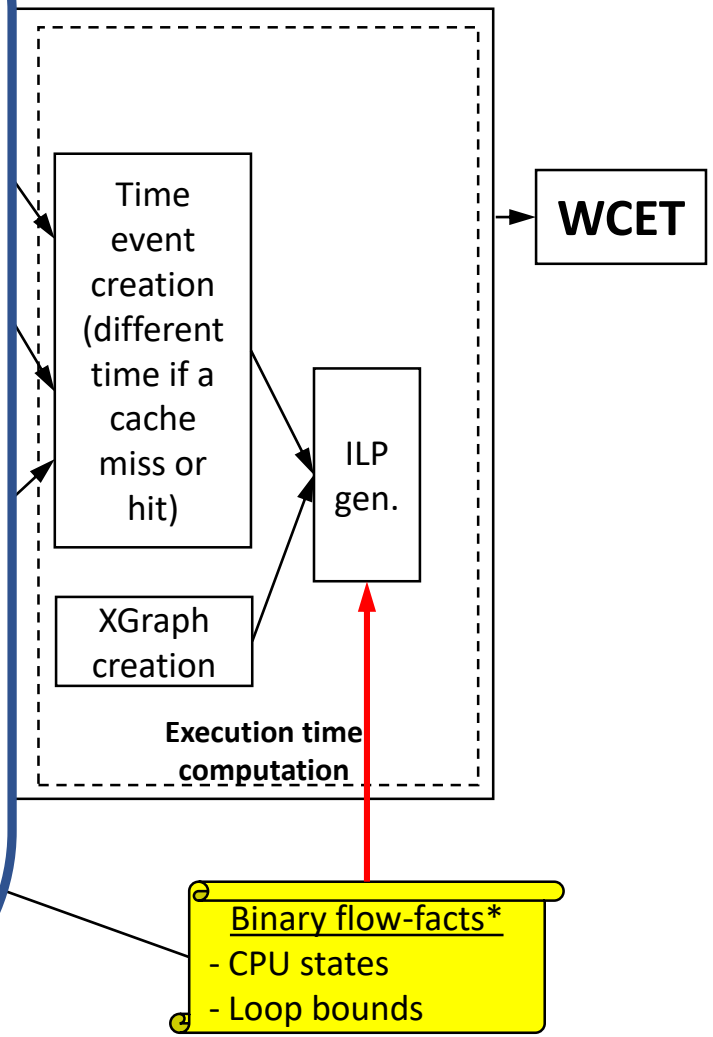


❖ Constrains as the upper bound of the loops into the formulae

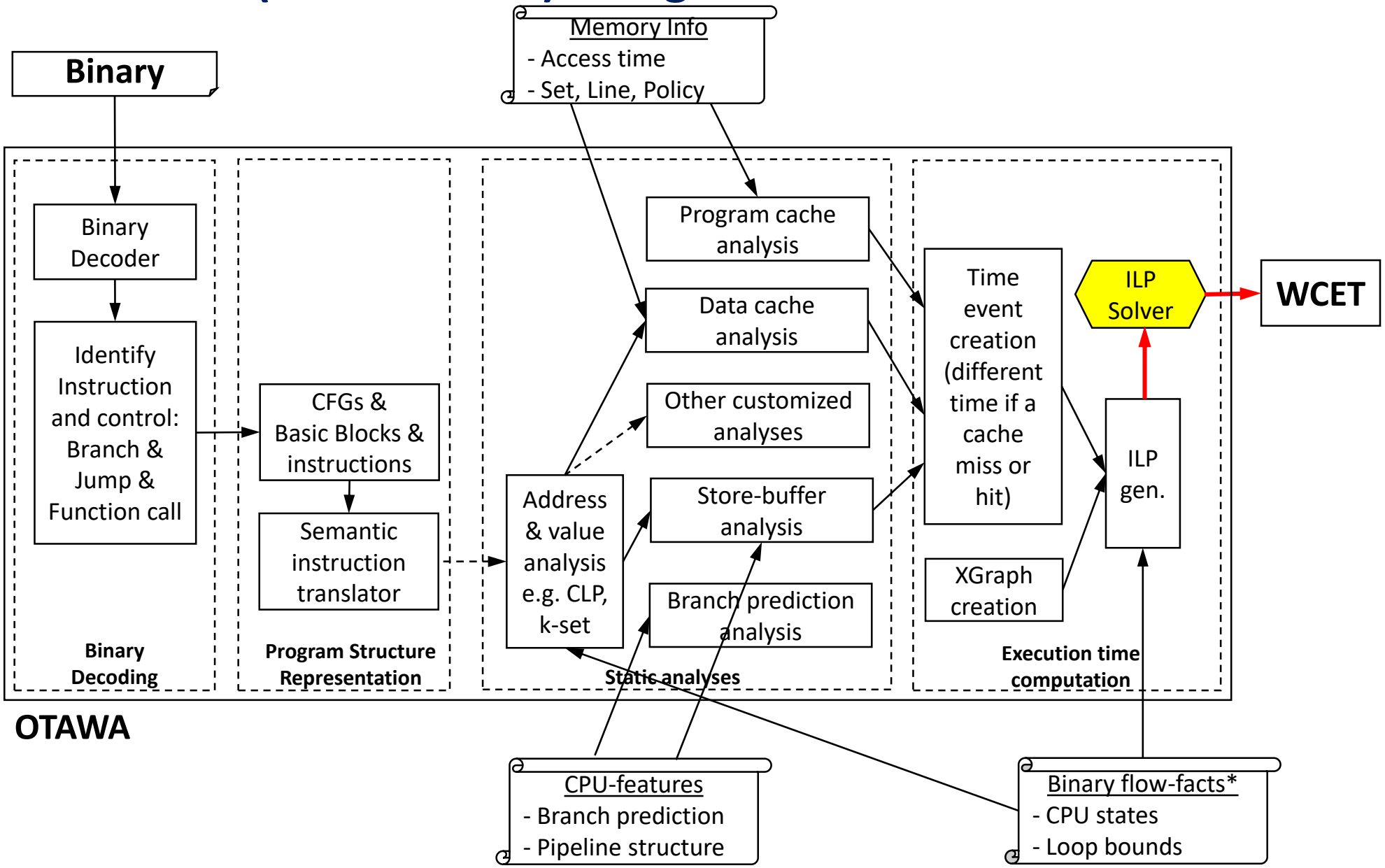


❖ Constrains as the upper bound of the loops into the formulae

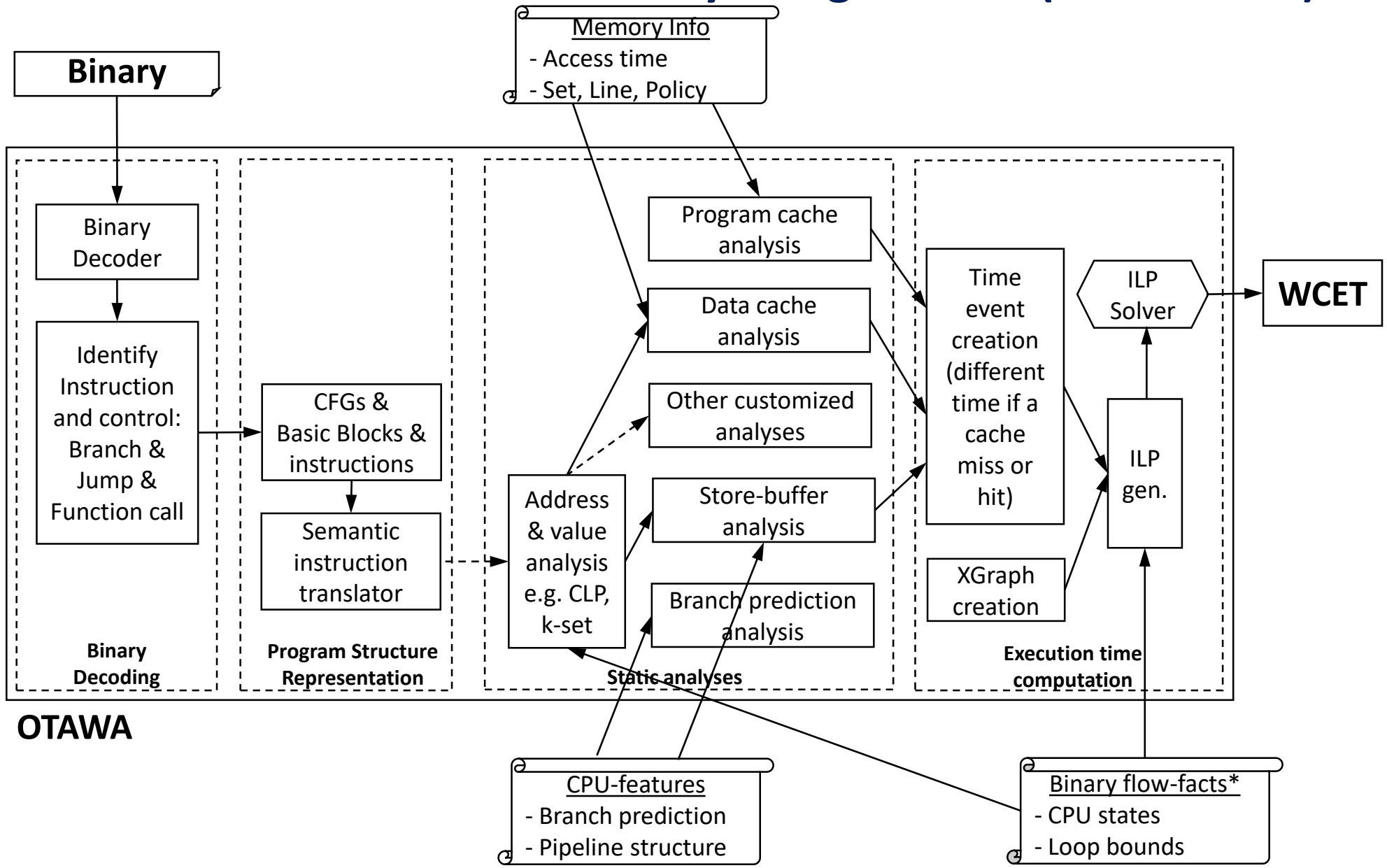
```
<flowfacts>
    <functionlabel=”binarysearch”>
        <loop label    = ”binarysearch”
              offset   = ”0xa0”
              maxcount = ”23” >
        </loop>
    </function>
</flowfacts>
```



❖ Solving the ILP formulae (via LP Solver) and get the WCET



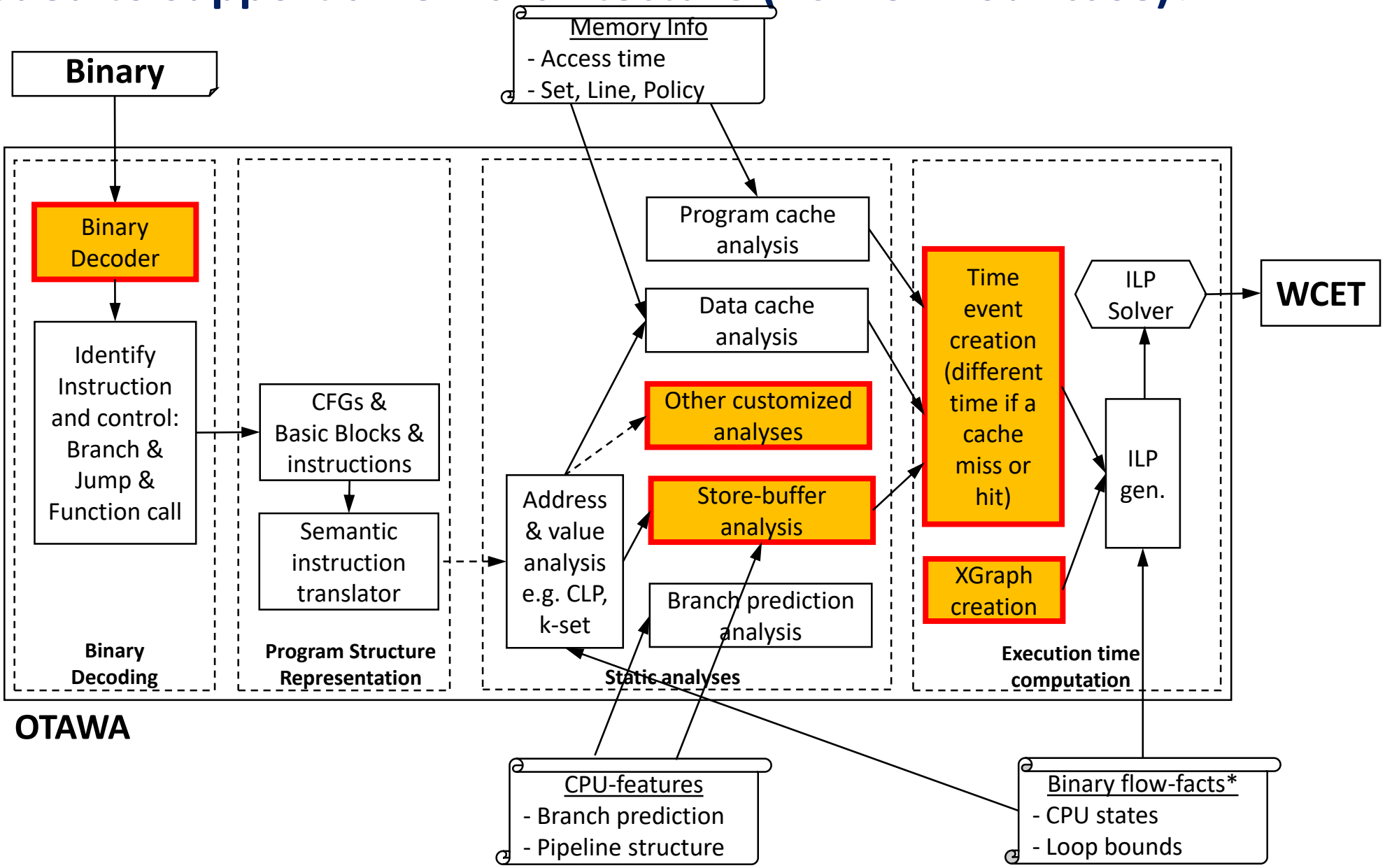
❖ The overview of the static WCET estimation by using OTAWA (User's view)



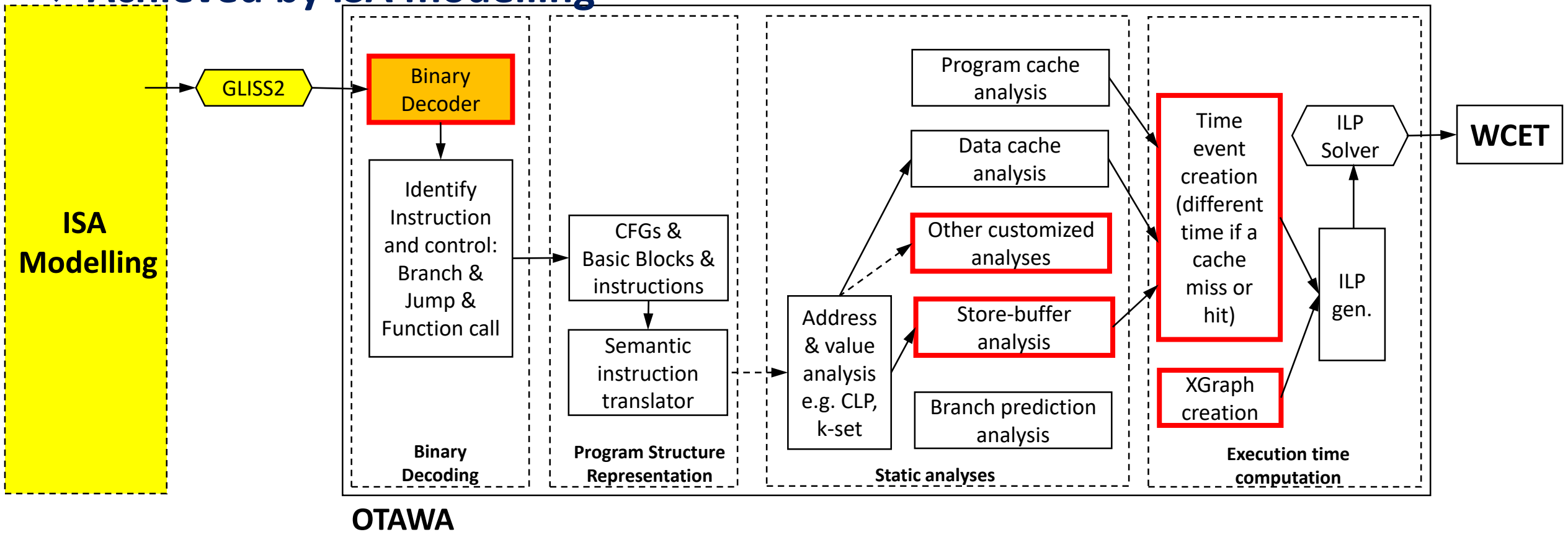
- User's perspective give us the idea what components are necessary
  - To compute WCET for an architecture
  - OTAWA provides a set of built-in analyses
  - Analyses are made platform-independent and can be configured
- To support OTAWA on a new platform, upon on the user's perspective
  - Introduce the **developer's perspective**
  - To create the OTAWA components which are platform-dependant

❖ What are needed to support a new architecture (TC275 in our case)?

Developers View

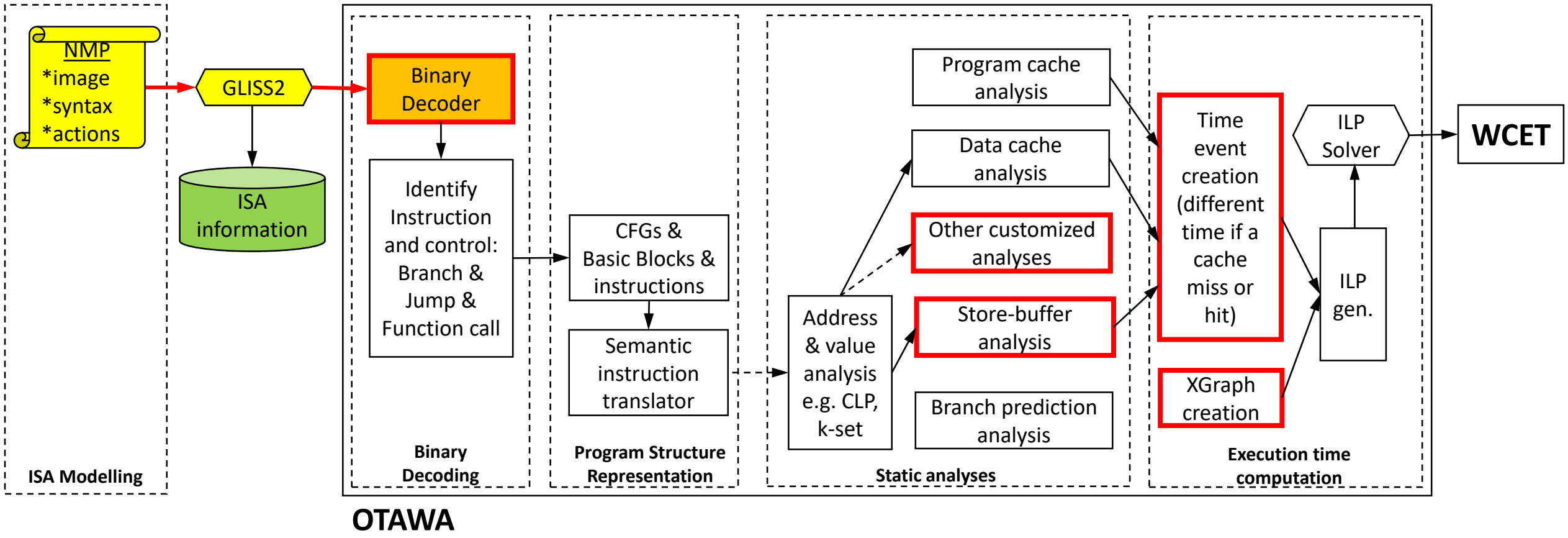


- ❖Developer's perspective – what needs to implement to support new arch.
- ❖Binary decoding – the first step to support a new architecture
- ❖ Achieved by ISA modelling

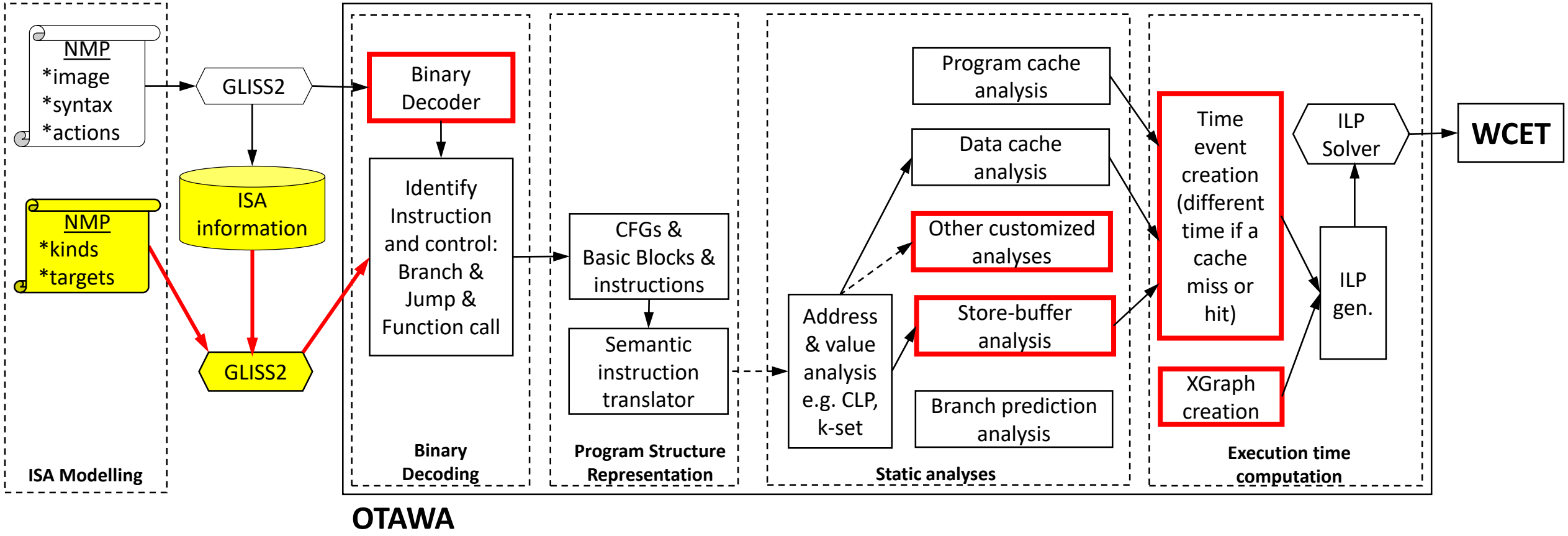


❖ Start with capture the ISA (Instruction Set Architecture)

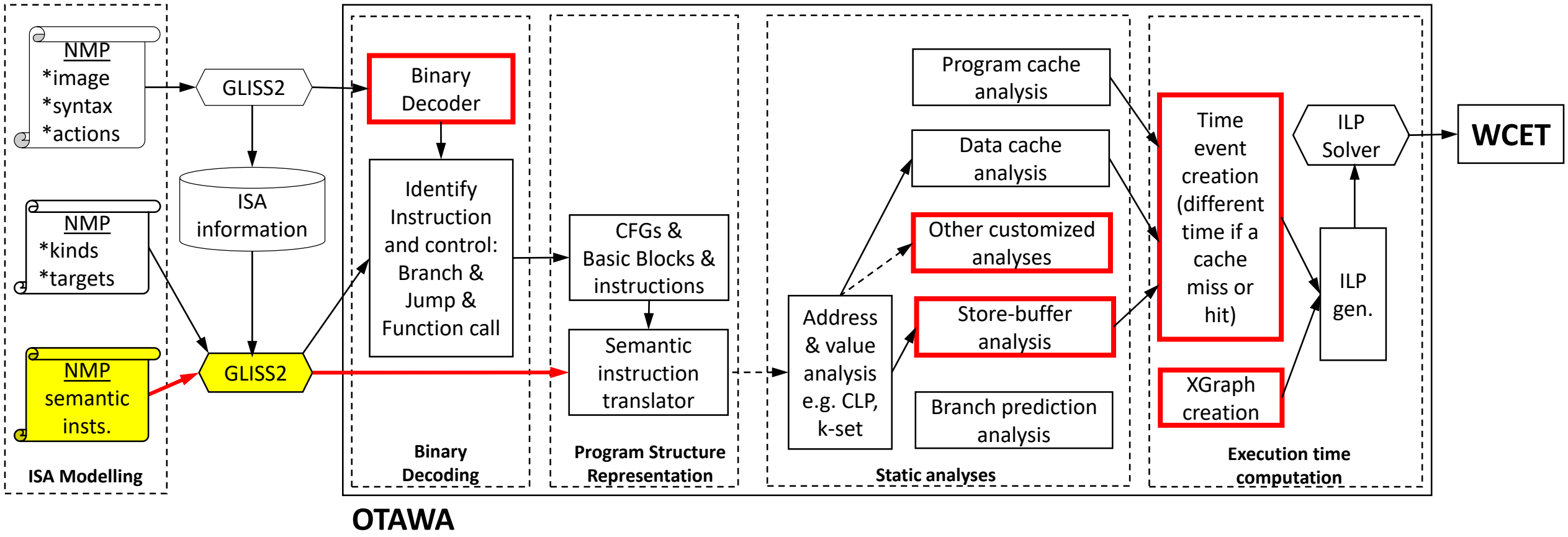
- ❖ Generate Binary decoder (by-product: ISS)
- ❖ Container that has all the ISA info



- Attributes can be associated with instructions
  - which are used to determine the boundaries of CFGs/BBs later
  - attributes are extensions to instructions that was previously stored in the container



- A set of semantic instructions is associated with each instruction
  - To describe its behaviour in a platform-independent manner
  - e.g. used by address & value analysis (CLP)



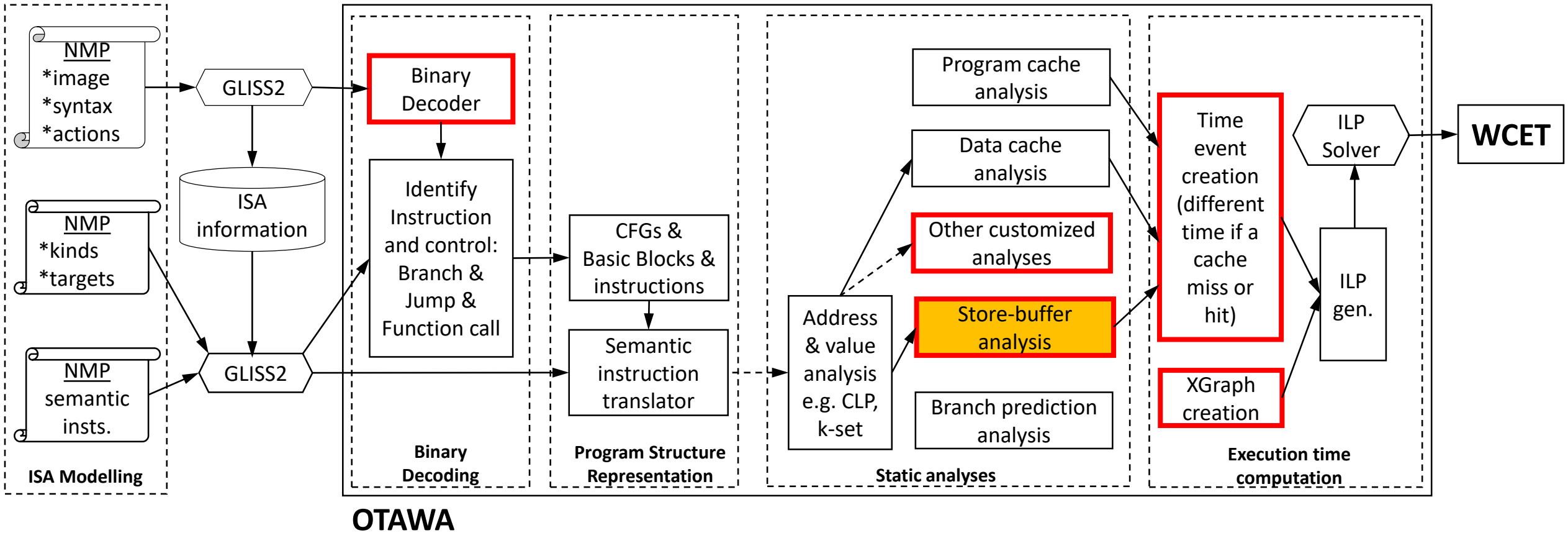
- How do we know if the ISA description that we obtained is correct?

- Validation of ISA description is performed
  - By checking with the official ISS (TSIM from Infineon for TC275)
  - With the generated ISS (act as binary decoder) from the ISA description
  - Published in WCET 2019
  - Present in the previous annual review

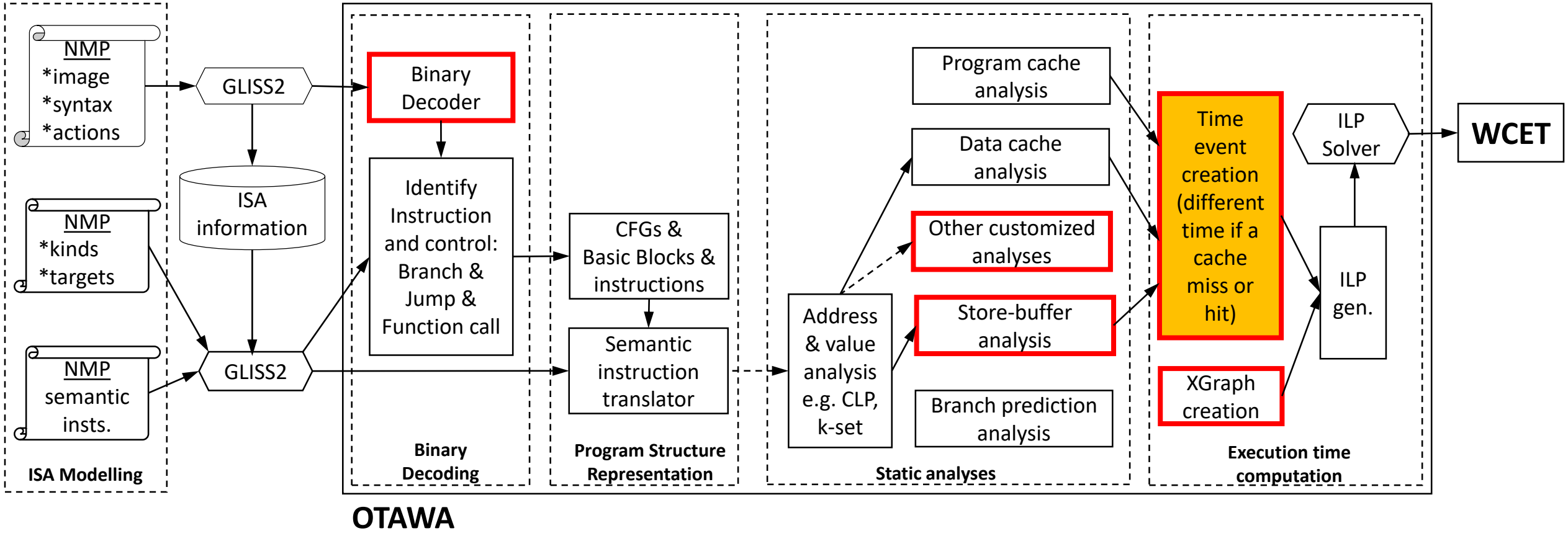
- Validation of the semantic instruction descriptions are performed also
  - By cross-checking if the abstract states cover the actual states
  - Abstract states obtained from static analyses
  - Actual states obtained from the generated ISS (verified in the previous step)
  - In the same paper of WCET 2019

❖ Some analyses can be adopted, adapted, and customized

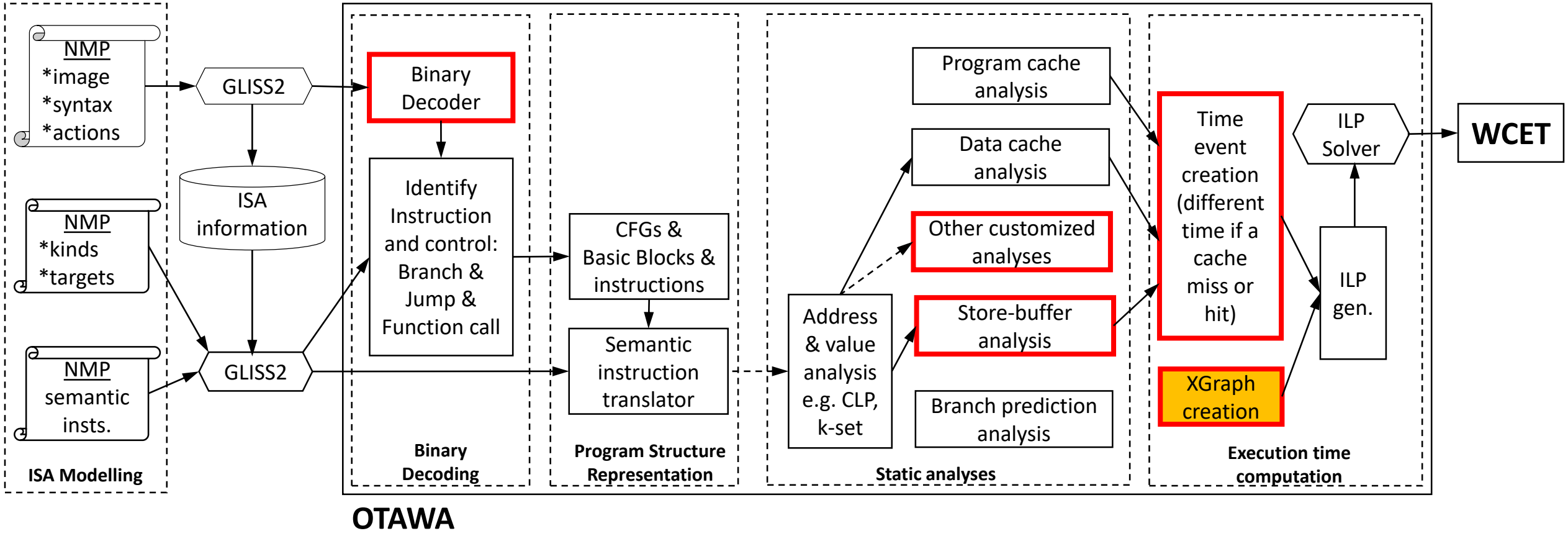
❖ Store-buffer analyses is similar of what present in ERTS 2018 (for Kalray Boston)



- Time events are to describes situation that takes extra time
  - besides the instruction timing in user-manual, e.g. cache-miss time
  - Time events depend on the analyses relevant to memory access in general

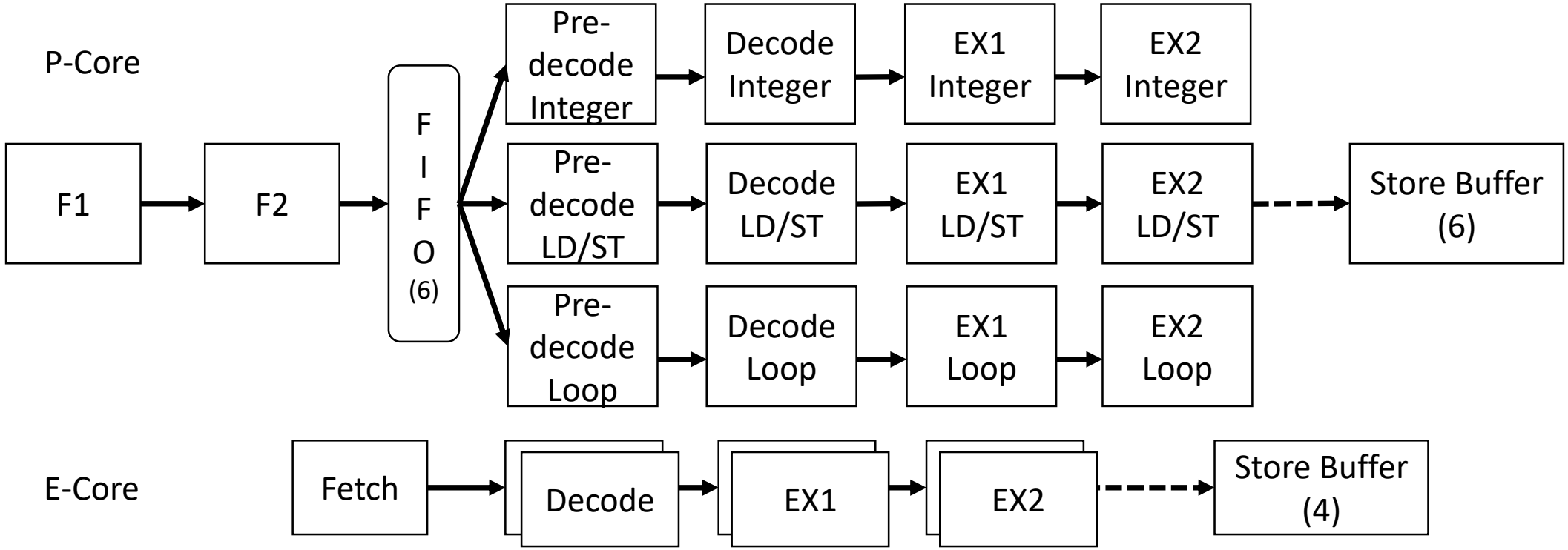


- Each eXcution Graph is used to compute the time for a instruction sequence
  - it also presents the dependencies of instructions during execution
  - Super-scalar effect in TC275's P-Core is captured in the XGraph



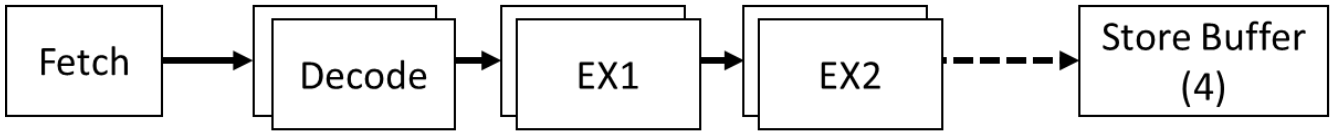
❖ Recall: the pipelines of TC275

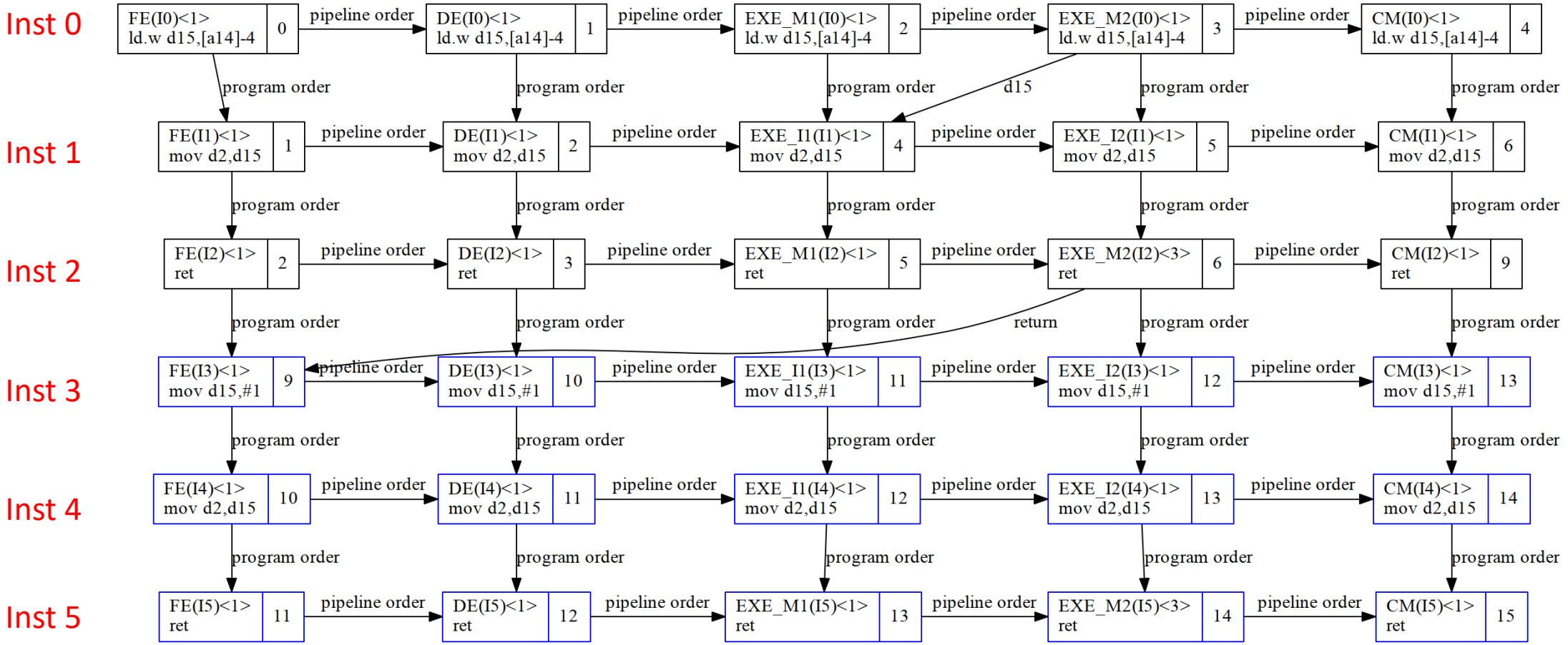
- ❖ Information obtained from user-manuals
- ❖ Some extra behaviours obtained from experiments



❖ eXcution Graph (XGtraph)

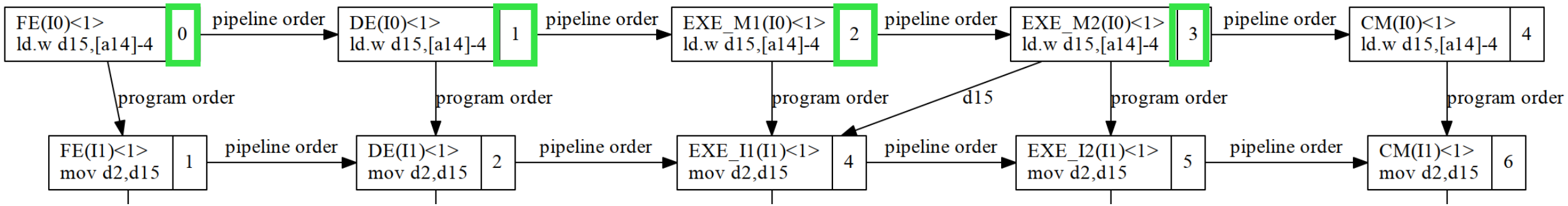
- ❖ capture the activities of instructions (vertical)
- ❖CPU's pipeline stages (horizontal)





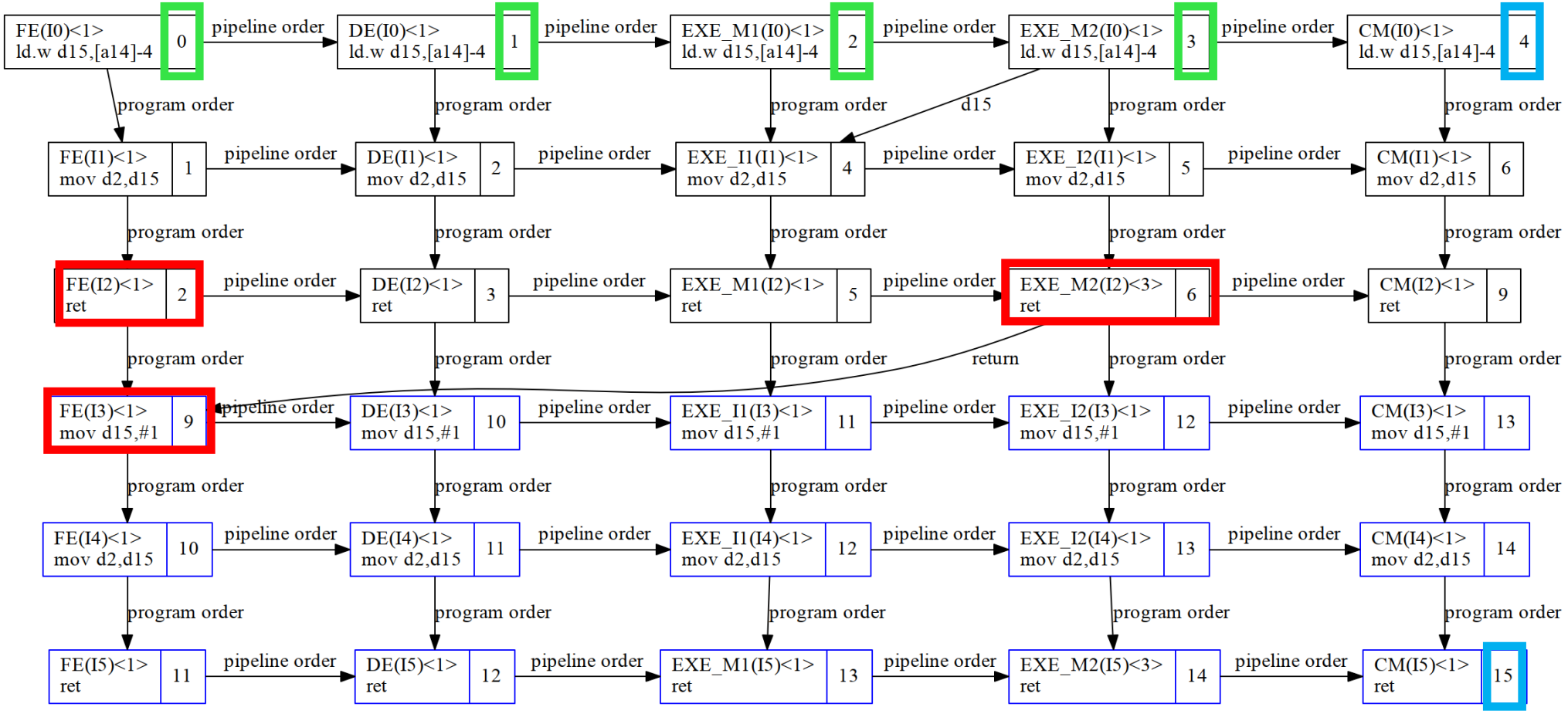
❖ eXcution Graph (XGtraph)

- ❖ The execution of each instruction takes some time at each stage
- ❖ Each node indicates the occurance of an instruction on a specific pipeline stage
- ❖ DE(I1)<1> – instruction 1 on the Decoding stage, which takes 1 cycle to finish
- ❖ The time in green boxes are the time of arrival
- ❖ Solid edges represent dependencies – target must wait all sources are finished



❖ eXcution Graph (XGtraph)

- ❖ The execution of each instruction takes some time at each stage
- ❖ The entry time of an instruction of a stage is the maximum time of the source nodes
- ❖ Time taken for the instruction sequence = end time of first and last instructions = 11

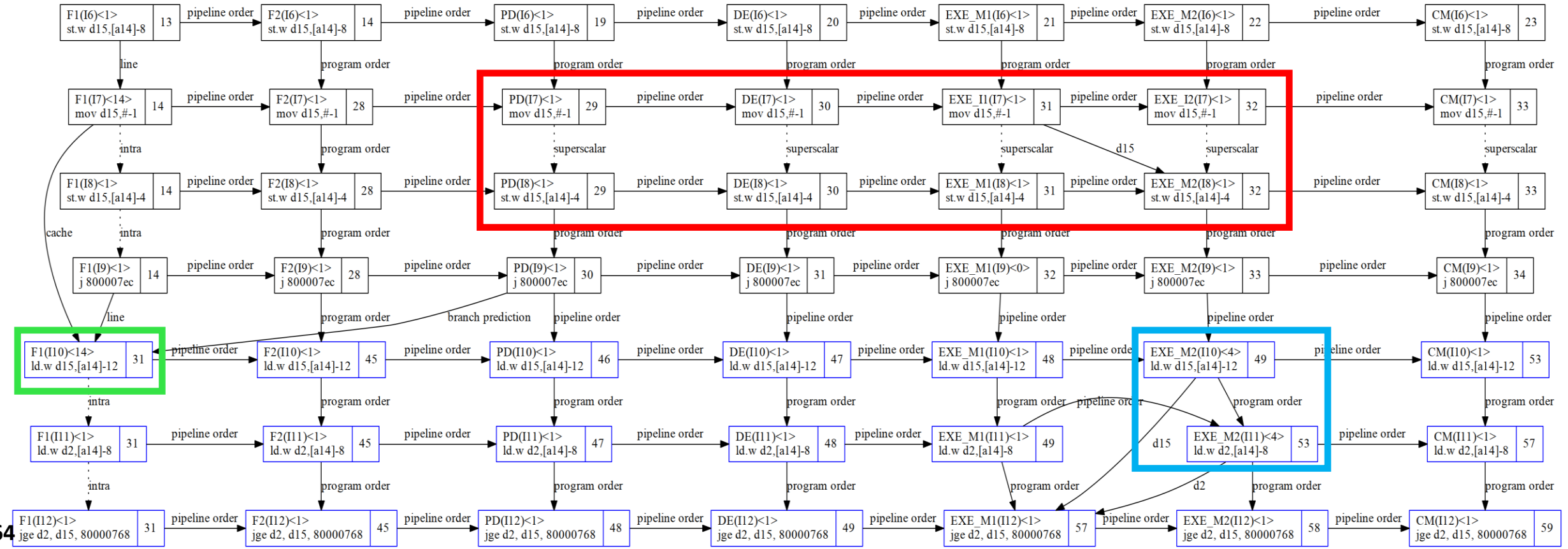


❖ Experiments super-scalar effect (instruction instrumenting)

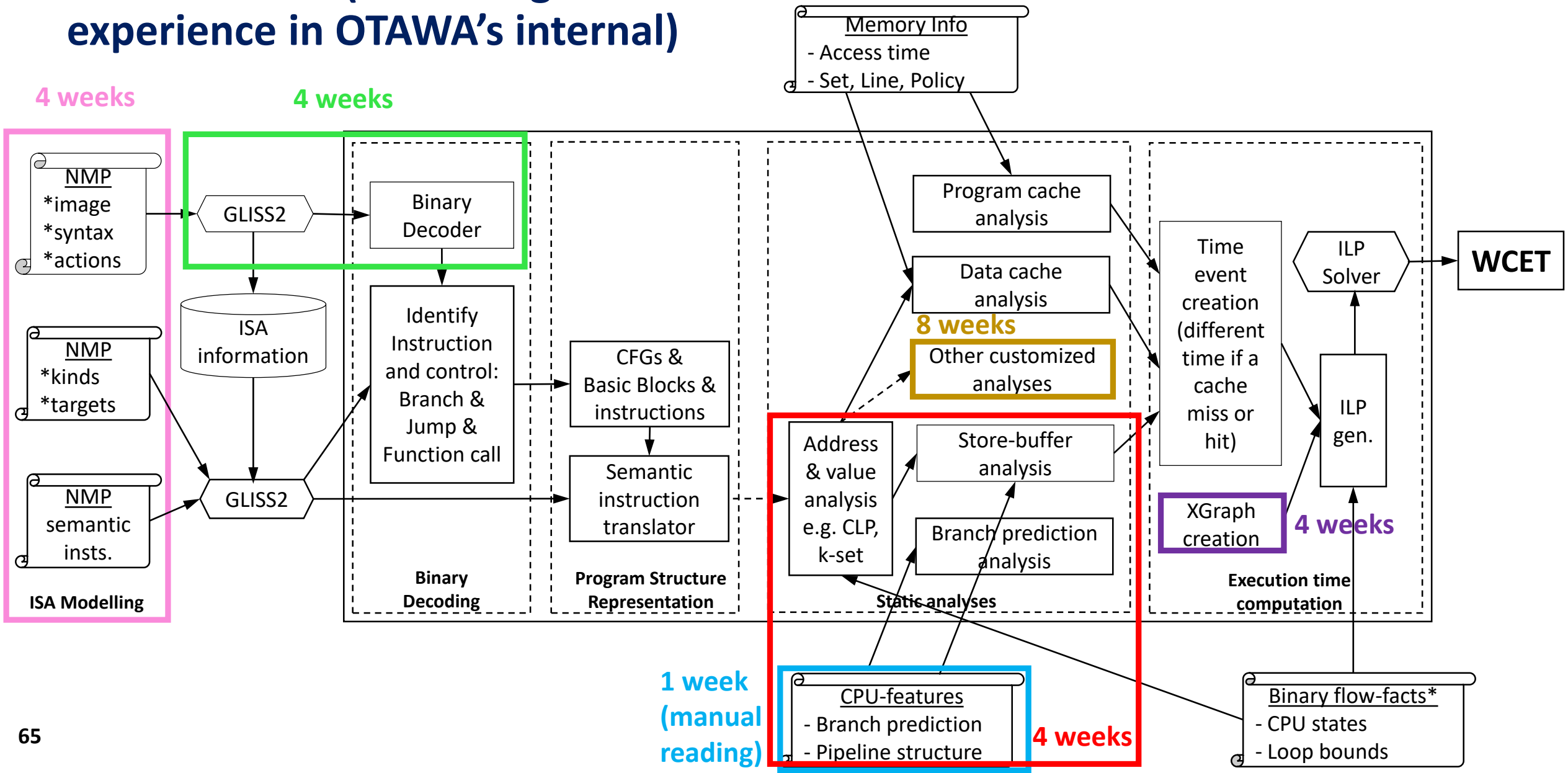
Seq	Address	Instruction						0	1	2	3	4	5
-1	0x800011bc	isync											
0	0x800011c0	mtcr 0xfc00,d15	Fetch...	0	PD.M	DE.M	E1.M	E2.M					
1	0x800011c4	mov d4,5		1	0	PD.I	DE.I	E1.I	E2.I				
2	0x800011c6	mov.aa a6,a5		2	1	PD.M	DE.M	E1.M	E2.M				
3	0x800011c8	mov d4,5			2	0	PD.I	DE.I	E1.I	E2.I			
4	0x800011ca	mov.aa a6,a5			3	1	PD.M	DE.M	E1.M	E2.M			
5	0x800011cc	mov d4,5			4	2	0	PD.I	DE.I	E1.I	E2.I		
6	0x800011ce	mov.aa a6,a5			5	3	1	PD.M	DE.M	E1.M	E2.M		
7	0x800011d0	mov d4,5				4	2	0	PD.I	DE.I	E1.I	E2.I	
8	0x800011d2	mov.aa a6,a5				5	3	1	PD.M	DE.M	E1.M	E2.M	
9	0x800011d4	mtcr 0xfc00,d2				w	4	2	0	PD.M	DE.M	E1.M	E2.M
10	0x800011d8	movh.a a15,0					5	3	1	0	PD.M	DE.M	E1.M
11	0x800011dc	movh.a a15,0					w	4	2	1	0	PD.M	DE.M
12	0x800011e0	movh.a a15,0						Fetch	-	-	-	-	0
	9 instructions	5 cycles											
	4	PMEM_STALL							1	2	3	4	
	0	DMEM_STALL											
	0	PCACHE_MISS											
	0	IP_STALL											
	0	LS_STALL											
	1	MULTI_ISSUE							1				
	0	DCACHE_HIT											
	0	DCACHE_MISS											
	0x800011C0	SRI_ACCESS											

❖ How other aspects of the execution are captured

- ❖ Super-scalar effect – 2nd inst. performs at the same time as the prev. (dash lines)
- ❖ Time event such as cache-miss contributes to the associated stage
  - ❖ instruction cache miss in green (effective on the Fetch stage F1)
  - ❖ memory access to LMU in blue (for load instructions, effective on stage EXE_M2)



❖ Efforts taken (for an engineer who is familiar with the HW and some experience in OTAWA's internal)



- Time and efforts for OTAWA to support TC275
  - For an engineer who is familiar with the HW and strong experience with OTAWA's design
  - Creation ISA descriptions in NMPs: 4 weeks
  - Validation of the ISA (generated ISS): 4 weeks
  - Configuration of the processor features
    - Finding the pipeline structure: 1 week (user-manual reading)
    - Instrumenting with instruction sequence: 4 weeks
  - Customisation of the XGraph: 4 weeks
  - Customisation of static analyses: 8 weeks
- Efforts for OTAWA to support FlexPRET: 3 days
  - Binary decoding of RISC-V already implemented in OTAWA

❖ The achievements

- ❖ Obtained WCET on TC275
- ❖ High support for E-core in particular
- ❖ Over-approximation for P-Core but safety is ensured
- ❖ Provide guide-line of support other platforms (ERTS 2020)
- ❖ Support of FlextPRET (easy thanks to its design)

❖ The remaining work

- ❖ Improve precision on P-core by modelling
  - ❖ Fetch-buffer to hide the penalty with cache-miss
  - ❖ Write-buffer used to decouple the memory access and other CPU ops.

❖ Lesson learnt

- **Information of the processors are hard to obtain**
  - Need experiments
  - Tracing tools
  - Vendor initiation – they know what's really inside the CPU

- **Open-source WCET framework such as OTAWA**
  - Easy to access
  - FREE!!
  - Require some experiences

Critical Applications
on Predictable High-Performance Computing Architectures

Thank you and Questions?

Don’t forget CAPHCA’s series of presentation in ERTS2

- **WE.2.C.3 @ 17h30** - Interferences in Time-Trigger Applications
- **TH.1.C.2 @ 11h45** - Model checking for timing interferences
- **FR.1.A.2 @ 09h30** - Design embedded SW in complex HW